



## behaviac 概述



# 目录

1 项目概况.....	5
1.1 behaviac 是我们对行为树（Behavior Tree）的一种实现方案 .....	5
1.2 主要特性及用户价值.....	5
1.3 项目链接.....	5
2 背景 .....	6
2.1 有限状态机（Finite State Machine，FSM） .....	6
2.2 行为树（Behavior Tree，BT） .....	7
2.3 一个例子.....	7
2.4 更多的说明.....	8
2.4.1 Log 文件 .....	9
2.4.2 一棵不好的行为树.....	10
3 Overview .....	10
4 元信息（meta），对 Agent 的描述.....	10
4.1 C++库中 Meta 的声明 .....	11
4.2 C#库中 Meta 的声明 .....	12
4.2.1 非 public 的 Method 和 Member.....	12
4.3 导出元信息.....	13
4.3.1 C++代码.....	13
4.3.2 C#代码.....	13
4.4 产生的 Meta 的 xml 文件 .....	14
5 工作区（workspace） .....	14
5.1 导出数据.....	14
5.2 导出路径.....	14
5.3 导出格式.....	16
5.4 命令行导出.....	17
6 编辑器（designer） .....	17
6.1 条件（Condition） .....	17
6.2 动作（Action） .....	18
6.3 注册 Method.....	18
6.4 子树 .....	19
6.4.1 递归.....	20
6.5 预制件（Prefabs） .....	20
6.6 par .....	21
6.7 主要节点类型.....	22
6.7.1 附件.....	22
6.7.2 叶子节点.....	22
6.7.3 分支节点.....	22
6.8 给节点配置 EnterAction/ExitAction.....	23
6.9 配置 Config.xml .....	23
7 事件处理.....	24
7.1 依靠 Parallel 节点.....	24
7.2 SelectorLoop 和 WithPrecondition .....	25

<https://github.com/TencentOpen/behaviac>

7.3 Event 作为附件.....	25
8 调试 .....	27
8.1 连接游戏.....	27
8.2 检查 BT 的运行情况.....	27
8.3 边框颜色.....	27
8.4 Agent::SetIdMask 和 Agent::SetIdFlag.....	28
8.5 查看或修改属性（property） .....	28
8.6 设置断点.....	28
8.7 时间轴（Timeline） .....	29
9 注册名字和实例.....	29
9.1 关于注册和绑定的说明.....	30
10 对 Agent 类型的指针的访问 .....	30
11 加载 .....	32
11.1 热加载 .....	32
11.2 更新包 .....	32
12 C++程序使用 behaviac .....	32
12.1 版本说明.....	33
12.2 开发版本和最终版本.....	33
12.3 主要步骤.....	34
12.4 几点特别注意.....	35
12.5 执行 BT，World::btexec .....	36
12.6 FileManager .....	37
12.7 关于 char，signed char，unsigned char 等的说明 .....	37
12.8 关于整合 behaviac 库的 FAQ.....	37
13 Unity 中使用 behaviac.....	38
13.1 使用 behaviac 需要的文件 .....	38
13.2 版本说明.....	39
13.3 Editor Script.....	40
13.4 Resouces .....	40
13.4.1 FileMansager .....	41
13.5 Logging 和 Connect（连接 Designer） .....	41
13.6 Script 及设置 .....	42
13.7 执行 BT .....	44
13.8 关于 GC ALLOC 的说明.....	44
14 关于 log 文件的 FAQ.....	44
15 和编辑器建立连接时的 FAQ.....	45
16 Spaceship 的例子 .....	46
17 主要节点的介绍.....	47
17.1 节点共有属性.....	48
17.1.1‘注释’和‘背景颜色’ .....	48
17.1.2 节点 ID .....	48
17.2 节点状态.....	48
17.3 组合节点 Composites.....	49
17.3.1 选择（Selector） .....	49

<https://github.com/TencentOpen/behaviac>

17.3.2 概率选择 (SelectorProbability)	50
17.3.3 随机选择 (SelectorStochastic)	51
17.3.4 序列 (Sequence)	51
17.3.5 随机序列 (SequenceStochastic)	52
17.3.6 条件执行 (IfElse)	53
17.3.7 并行 (Parallel)	53
17.3.8 循环选择 (SelectorLoop) 和条件动作 (WithPrecondition)	54
17.4 叶子节点	55
17.4.1 动作 (Action)	55
17.4.2 赋值 (Assignment)	56
17.4.3 计算 (Compute)	56
17.4.4 等待 (Wait)	57
17.4.5 等待帧数 (WaitFrames)	57
17.4.6 空操作 (Noop)	57
17.4.7 查询 (Query)	57
17.4.8 等待信号 (WaitforSignal)	60
17.5 条件节点	60
17.5.1 条件 (Condition)	60
17.5.2 或 (Or) 及与 (And)	61
17.5.3 真 (True) 及假 (False)	61
17.6 装饰节点	61
17.6.1 输出消息 (Log)	61
17.6.2 非 (Not)	62
17.6.3 循环 (Loop)	62
17.6.4 循环直到 (LoopUntil)	62
17.6.5 计数限制 (CountLimit)	62
17.6.6 返回成功直到 (FailureUntil) / 返回失败直到 (SuccessUntil)	63
17.6.7 总是成功 (AlwaysSuccess) / 总是失败 (AlwaysFailure) / 总是运行 (AlwaysRunning)	63
17.6.8 时间 (Time) 及帧数 (Frames)	64
17.7 附件	64
17.7.1 判断 (Predicate)	64
17.7.2 事件 (Event)	65
18 Revision	65

# 1 项目概况

## 1.1 behaviac 是我们对行为树（Behavior Tree）的一种实现方案

- 已陆续开发 2 年，成熟稳定
- 久经考验，多个项目使用
- 全平台支持：windows、linux、android、ios 等
- 对 Unity 有 C# 的原生支持
- 中英文界面，文档丰富，多个 tutorials

## 1.2 主要特性及用户价值

- 编辑器和 C++/C# 的交互基于元信息（property 和 method），既能充分利用程序代码的各种功能，也提供了图形化的对于高层逻辑的形象化控制
- 通过 par，具有类型的命名的变量，外部系统可以和 BT 交换信息，这使得整个系统有了动态的能力
- xml、bson 等导出格式，还有 Cpp 或 C# 文件导出，既提供了加载、执行的高效，也使用热加载极大的提高了开发效率。
- 功能完善易用的编辑器，prefab、undo、子树、事件等的支持
- 支持的数据类型和节点类型可方便的扩展
- 实时或者离线调试，使得运行逻辑图形化，具体化，方便调试
- 使用场景，不只是 AI，支持并不限于
  - ◇ Character AI
  - ◇ Squad Logic
  - ◇ Strategy AI
  - ◇ In-Game Tutor
  - ◇ Animation Control
  - ◇ Player Avatars

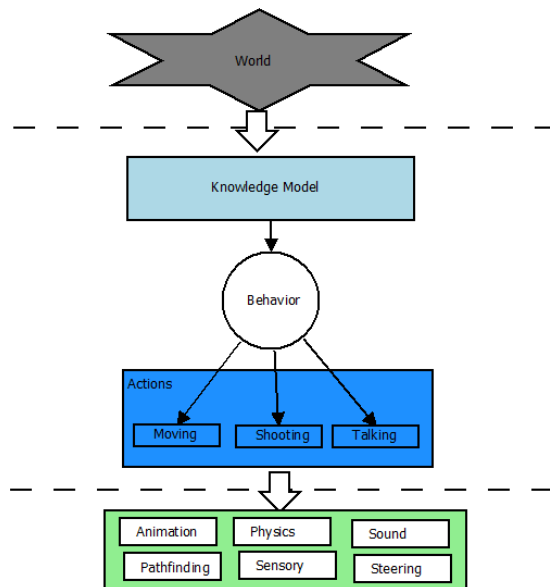
## 1.3 项目链接

可以参考下面网站获取最新版本等更多信息：

<https://github.com/TencentOpen/behaviac>

## 2 背景

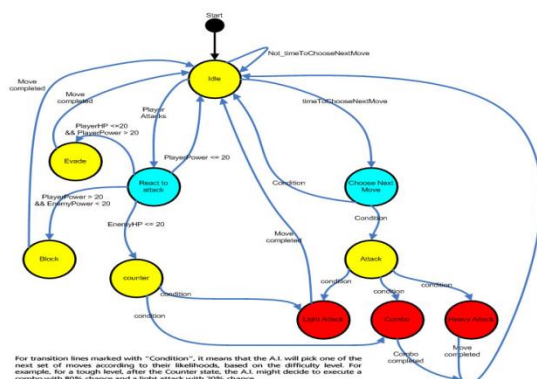
游戏 AI 的目标之一就是要找到一个**简单**并且可**扩展**的编辑逻辑（logic）的方案。  
行为系统（Behavior System）响应游戏中的各种信息，进行决策以挑选接下来执行的行动并且监控行动的执行。



知识模型（Knowledge model）是对游戏世界中各种信息的抽象。如何高效和方便的表示知识模型（KB）至关重要。

### 2.1 有限状态机（Finite State Machine, FSM）

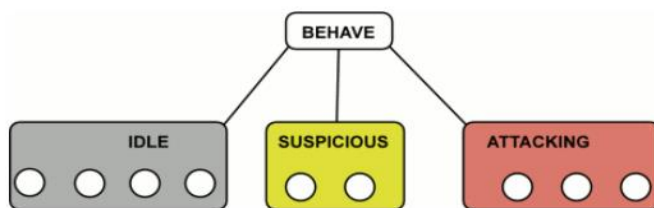
FSM(finite state machine)有着简单的优势，对于大型的系统，HFSM(hierarchical FSM)也支持了状态间的切换的重用。但是 FSMs 需要用转换（transition）连接状态（State），从而状态（State）失去了模块性（modularity）。



## 2.2 行为树（Behavior Tree, BT）

BT 的方案中，每个节点的执行结果（success/failure/running）都有其父节点来管理，从而决定接下来做什么，父节点的类型决定了不同的控制类型。节点不需要维护向其他节点的转换，节点的模块性（modularity）被大大增强了。实际上，在 BT 里，由于节点不再有转换，他们不再是状态（State），节点只是行为（Behavior）。

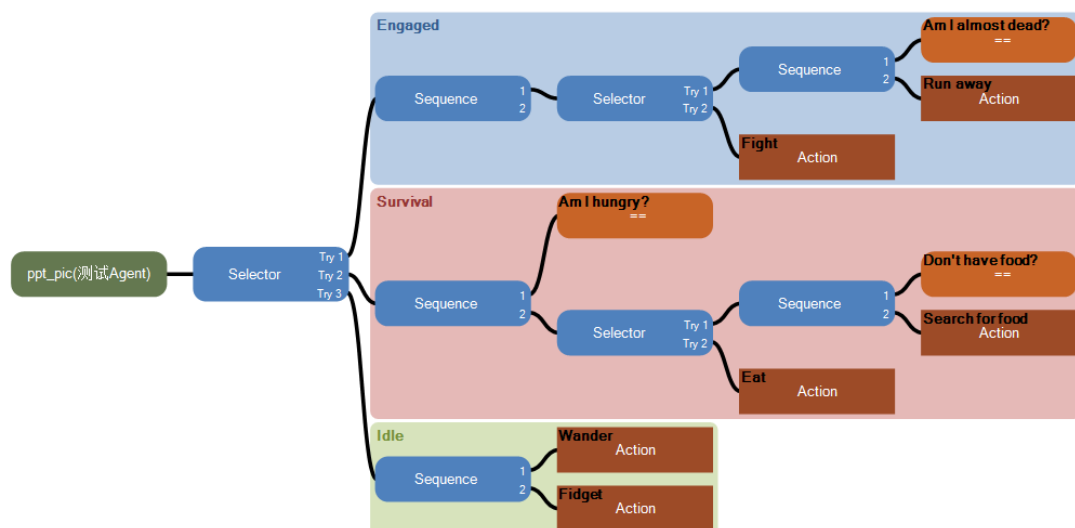
概念上，一个 BT 就是一段脚本（script）或程序代码（code），只不过是树（tree）的形式而不是以线性代码的形式表现给用户。



## 2.3 一个例子

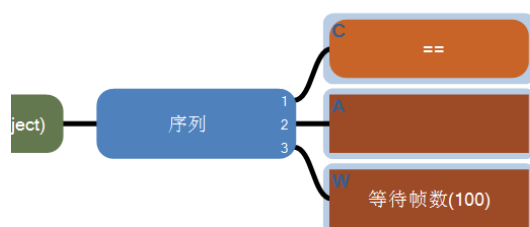
一个 BT 被如下组织从而实际上你在按照一定的优先级问问题，然后执行行动：

- Am I being attacked? [Am I almost dead? Run away!] [Fight!]
- Am I hungry? [Don't have food? Search for it.] [Eat]
- Idle [Wander] [Fidget]



## 2.4 更多的说明

然而若仔细思考 BT 的执行过程，还是有些疑问需要澄清。假若某个 BT 如下图，一个序列有 3 个子节点，分别是一个条件节点 C，一个动作节点 A，一个等待节点 W。



条件节点对左右参数进行比较，

- 如果比较结果为 True，则条件节点返回成功 Success
- 如果比较结果为 False，则条件节点返回成功 Failure
- 条件节点不可能返回 Running

而动作节点则需要根据配置或实现来决定其返回值。

假若动作节点 A 执行后直接返回成功（请参考[注册 Method](#)），而等待节点 W 是要等待 100 帧。C++代码调用 Agent::btexec 更新该 BT 的时候，如果代码类似下面的：

```
int frames = 1;
while (true) {
    BEHAVIAC_LOGINFO("Frame %d: behaviac::Agent::btexec", frames++);
    if (pAgent->btexec() != BT_RUNNING) {
        BEHAVIAC_LOGINFO("Finish");
        break;
    }
}
```

其执行过程则如下：

```
Frame 1: behaviac::Agent::btexec
    exec C
    exec A
    exec W

Frame 2: behaviac::Agent::btexec
    exec W

Frame 3: behaviac::Agent::btexec
    exec W

...
Frame 100: behaviac::Agent::btexec
    exec W

Finish
```

<https://github.com/TencentOpen/behaviac>



## 2.4.1 Log 文件

实际上，在 behaviac 中，当游戏运行时可以查看 exe 所在目录的 \_behaviac\_\$\_.log，该文件类似如下图：

```
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:exit [success] [1]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Selector[1]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Selector[1]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:exit [failure] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:exit [failure] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->ReferencedBehavior[29]:enter [success] [1]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->ReferencedBehavior[29]:update [running] [1]
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 bool par_alive->False
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_enemy->0
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_target->1
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 string par_category->
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_event_param->0
```

每一行的具体格式首先是用 ‘#’ 分割的类名和实例名。

由于对效率的考虑，缺省情况下，logging 文件没有激活，也许你不能发现文件 \_behaviac\_\$\_.log，请参考[关于 log 文件的 FAQ](#)

需要说明的是，此外还有另一个 log 文件（c#下没有这个文件）：\_behaviac\_\$\$\_log，该文件是包含 behaviac 运行过程中的 info，warning 或 error 或其他信息的，不是包含 BT 执行情况的，BT 的执行情况统一输出到 \_behaviac\_\$\_.log。

此外，还可以调用 `SetLogFilePath` 设置 Log 文件的文件名及目录。

### 2.4.1.1 执行状态行

[tick]行是 BT 的执行情况，某个 Agent 实例上的某个 BT 上的某个节点的执行情况。

[tick]ClassFullName#InstanceName BTName->NodeClassName[NodeId]:ExecAction [EBTStatus] [Count]

在用 ‘#’ 分割的类名和实例名后，是 BT 树，接下来用 ‘->’ 作为分割，接下来是节点的类型名字及用 ‘[]’ 括起来的节点 ID，接下来是用 ‘:’ 作为分割，之后是执行动作及用 ‘[]’ 括起来的执行状态。最后是用 ‘[]’ 括起来的数字表明执行次数。

### 2.4.1.2 属性行

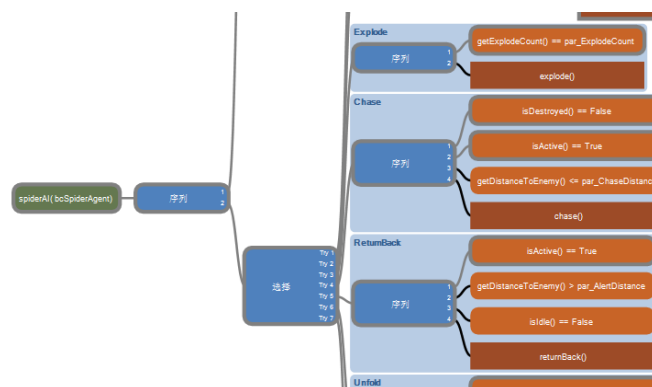
[property]行是某个 Agent 实例上的属性或 par 的值。

[property]ClassFullName#InstanceName Type Name->Value

在用 ‘#’ 分割的类名和实例名后，是数据类型，接下来用 ‘->’ 作为分割，接下来值。

## 2.4.2 一棵不好的行为树

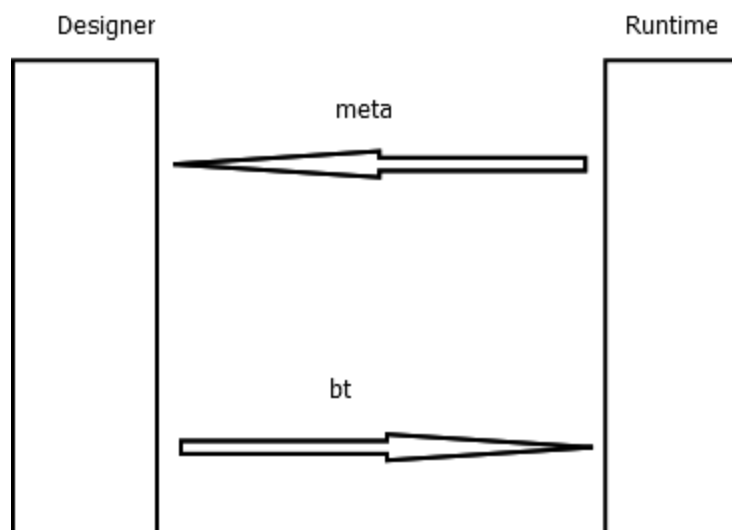
下图所示 BT 某种意义上不是一颗“好”的 BT。该 BT 每次执行要么返回成功，要么返回失败。而一个“好”的 BT 在大部分情况下应该返回运行（Running），返回成功或失败意味着该 BT 结束了。所以很多情况下需要在 BT 里添加一些类似循环的节点。



这样做的原因在于运行（Running）状态的节点在下次执行的时候会被直接执行，而一个返回成功或失败也就是结束了的 BT，下次执行的时候还必须重新从头开始。

## 3 Overview

behaviac 是我们对行为树 (Behavior Tree) 的一个实现方案。该方案包括一个编辑器 (Designer) 和一个 Runtime 的库 (c++/C#)。编辑器 (designer) 用来编辑和调试 BT，Runtime 的库用来解释和执行编辑器 (Designer) 生成的 BT。



## 4 元信息（meta），对 Agent 的描述

游戏中负责运行行为的实体称谓 Agent，每一个 Agent 都存有一定的数据或者能执行一定的

<https://github.com/TencentOpen/behaviac>

动作。这样的数据被称谓属性（property），这样的动作被称谓方法（method），所谓的元信息（meta）就是对一个 Agent 的描述，包括其属性（property）和方法（method）。Runtime 的库（C++ lib）或 C#的库产生一个描述 agent 元信息（meta）的 xml 文件。

需要指出和强调的是，元信息 meta 由 Runtime 生成，Designer 只是读入并且利用该元信息来‘描述’BT，元信息不是由 Designer 生成的。当然某些情况下也可以手工的修改该元信息来包括一些 Runtime 还没有实现的一些属性或方法从而做一些实验。

元信息是**仅供 Designer** 使用的，Runtime 不需要使用元信息，元信息只需要开发版本中当 Agent 有改变的时候导出一次就可以，不需要每次都导出。最终的发布版本中不需要导出元信息 meta 文件。

然而在开发版本中，既可以在一个特殊的工具中导出元信息，如 [Editor Script](#)，也可以每次运行开发版游戏的时候导出元信息(元信息文件是可写的)，这样子可以确保任何 Agent 的改动都可以更新到元信息 meta 文件中；另一方面，只要 Agent 没有改变，导出的元信息 meta 文件也将是一样的。

另外，如果是通过运行开发版游戏产生或者更新导出元信息，当修改了 Agent 后，原本根据旧的元信息产生的 BT 就不能运行了，有人会抱怨由于 BT 不能运行导致开发版的游戏 crash 而不能产生元信息。解决方法在于**不用纠结于游戏的 crash**，只要把产生元信息的函数调用放在使用 BT 前就可以了，因为产生元信息只依赖 Agent 的注册，不依赖于 BT。当然也可以考虑添加某个参数专门用来产生或更新元信息。

## 4.1 C++库中 Meta 的声明

在 C++的代码里，用户可以使用提供的 REGISTER\_PROPERTY 来声明属性（property），使用 REGISTER\_METHOD 来声明方法（Method），使用 REGISTER\_EVENT 来声明事件（event）。

```
BEGIN_PROPERTIES_DESCRIPTION(BtSnowNinja)
    REGISTER_PROPERTY(targetBrain).SETNETROLE(behaviac::NET_ROLE_NONAUTHORITY);

    REGISTER_METHOD(ClearFireJumpFlag);
    REGISTER_METHOD(FindTarget).ADDPARAM("type").ADDPARAM("alertRange");
    REGISTER_METHOD(DistanceTo).ADDPARAM("target");
    REGISTER_METHOD(Fire).ADDPARAM("target").ADDPARAM("fireRange").ADDPARAM("fire_scale");
    REGISTER_METHOD(Aim).ADDPARAM("target");
    REGISTER_METHOD(Move).ADDPARAM("target").ADDPARAM("farRange").ADDPARAM("closeRange");
    REGISTER_METHOD(RandomJump).ADDPARAM("jumpRange");
    REGISTER_METHOD(Wander);

    REGISTER_METHOD(Health);
    REGISTER_METHOD(SetDebugHealth).SETNETROLE(behaviac::NET_ROLE_NONAUTHORITY);

    REGISTER_METHOD(GetCollidedObject);
    REGISTER_METHOD(Heal);
    REGISTER_METHOD(TakeDamage);

    REGISTER_EVENT("Collided");
END_PROPERTIES_DESCRIPTION()
```

## 4.2 C#库中 Meta 的声明

```
//we declare a type "Float2" to simulate a type defined in a thirdparty lib
namespace TestNS
{
    [behaviac.TypeMetaInfo()]
    public struct Float2
    {
        public float x;
        public float y;
    }
}

namespace test_ns
{
    [behaviac.TypeMetaInfo()]
    public struct Param_t
    {
        public TypeTest_t type;
        public int color;
        public string id;
    }

    [behaviac.TypeMetaInfo("Agent基类 用来显示用的", "Agent基类的说明")]
    public class AgentBase : behaviac.Agent
    {
        [behaviac.MethodMetaInfo()]
        public behaviac.EBTStatus action0()
        {
            return behaviac.EBTStatus.BT_SUCCESS;
        }

        [behaviac.EventMetaInfo()]
        delegate bool EventExploded(TypeTest_t param);

        [behaviac.MemberMetaInfo()]
        public int Base_Property1;

        [behaviac.MemberMetaInfo()]
        private bool Base_Property2;
    }
}
```



具体的通过下面的几个 Attribute 来修饰相应的类型 class/struct, 属性 property, 方法 method, delegate, 参数等来表明该类型 class/struct, 属性 property, 方法 method, delegate, 参数等是需要导出的 meta 信息并且可以提供 DisplayName, Description 等信息:

- TypeMetaInfoAttribute
- MemberMetaInfoAttribute
- MethodMetaInfoAttribute
- EventMetaInfoAttribute
- ParamMetaInfoAttribute

### 4.2.1 非 public 的 Method 和 Member

导出的BT可以是C#源码。当Method和Member不是public的时候, 在导出的C#源码里访问该Method和Member的代码就只能通过C#的Reflection系统来进行, 其效率不是最优的。而当它们是public的时候则可以直接访问, 而没有多余的overhead。

缺省情况下, 非public的Method和Member被导出, 但是在Console窗口输出warning:

 Export non-public method : GameLevelCommon.stopGame  
UnityEngine.Debug.LogWarning(Object)  
 Export non-public method : GameLevelCommon.stopGame  
UnityEngine.Debug.LogWarning(Object)

## 4.3 导出元信息

具体的通过 `Workspace::ExportMetas` 来导出元文件。调用 `Workspace::ExportMetas` 来导出元文件的时候，可以指定第二个参数 `onlyExportPublicMembers=true` 来指定只导出 `public` 的 `Method` 和 `Members`。

此外，可以通过 `Config.IsSupressingNonPublicWarning=true` 来不输出 `warning`。

### 4.3.1 C++代码

```
//register agents
behaviac::Agent::Register<TestAgent>();

//register names
Agent::RegisterName<Agent>("AnyAgent");

//after all the 'Agent::Register', export meta info,
//this is only needed to execute when you Agent's structure is modified
behaviac::Workspace::ExportMetas("agents\\testAgents.xml");
```

1. 注册 `Agent` 类型。
2. 注册名字，参考[注册名字和实例](#)，只是导出元信息是不需要绑定实例的。
3. 调用 `ExportMetas` 导出元信息到指定的文件。

### 4.3.2 C#代码

```
//register agents
behaviac.Workspace.RegisterMetas ();

//register names
behaviac.Agent.RegisterName<GameLevelCommon> ("GameLevel");

string metaExportPath = GameLevelCommon.WorkspacePath + "/xmlmeta/BattleCityMeta.xml";
behaviac.Workspace.ExportMetas (metaExportPath);
behaviac.Debug.Log ("Behaviac meta data export over.");
```

1. 注册 `Agent` 类型。与 `C++`不同的是不需要逐个的注册每一个 `Agent`，`RegisterMetas` 函数自动遍历所有被 `TypeMetaInfo` 修饰的 `Agent` 子类并且注册之。
2. 注册名字，参考[注册名字和实例](#)，只是导出元信息是不需要绑定实例的。
3. 调用 `ExportMetas` 导出元信息到指定的文件。

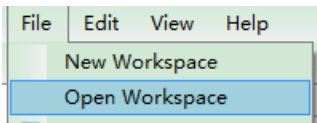
## 4.4 产生的 Meta 的 xml 文件

```
<metas>
  <agents>
    <agent classfullname="behaviac::World" inherited="true" DisplayName="" Desc="" />
    <agent classfullname="framework::GameObject" inherited="true" DisplayName="" Desc="">
      <Member Name="HP" DisplayName="" Desc="" Type="unsigned int" Class="framework::GameObject" />
      <Member Name="age" DisplayName="" Desc="" Type="signed long" Class="framework::GameObject" />
      <Method Name="GoStraight" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="void">
        <Param DisplayName="speed" Desc="speed" Type="signed int" />
      </Method>
      <Method Name="TurnTowardsTarget" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="signed int">
        <Param DisplayName="turnSpeed" Desc="turnSpeed" Type="float" />
      </Method>
      <Method Name="alignedWithPlayer" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="bool" />
      <Method Name="playerIsAligned" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="bool" />
      <Method Name="projectileNearby" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="bool">
        <Param DisplayName="radius" Desc="radius" Type="float" />
      </Method>
      <Method Name="distanceToPlayer" DisplayName="" Desc="" Class="framework::GameObject" ReturnType="float" />
    </agent>
    <agent classfullname="framework::Projectile" base="framework::GameObject" DisplayName="" Desc="" />
  </agents>
</metas>
```

Runtime，无论是 C++ 还是 C# 导出的元信息 Meta 文件的格式都是一样的。

## 5 工作区（workspace）

工作区(workspace)文件是一个管理一个项目的配置文件。该配置文件可以指定元信息(meta)的 xml 文件，Behavior 的源文件路径，导出文件的路径等。可以从 File 菜单下选择创建、打开或编辑一个 workspace。



Behavior 的源文件是指 BT 的 xml 文件，该 xml 文件只在 Designer 中使用。源文件必须导出后才能在 Runtime 中使用。导出的格式可以是 xml, bson, cpp, c# 等。导出文件将被导出到‘导出路径’中。

源文件就像那些 RawData，而导出文件就像那些处理过在游戏中直接使用的 GameData。

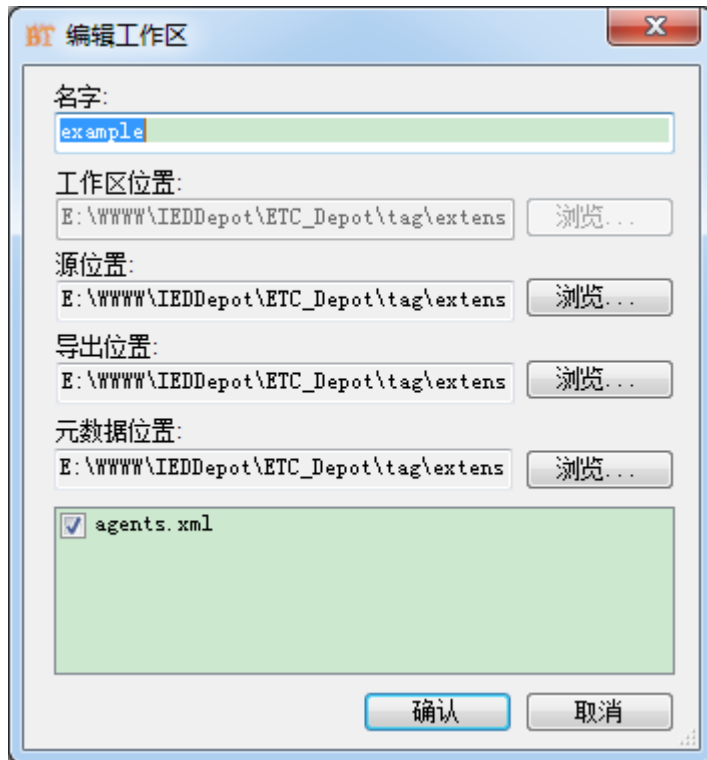
### 5.1 导出数据

编辑器中可以创建和修改 BT，游戏使用 BT 的时候，需要在编辑器中导出（export）。工作区文件，源文件，元信息 meta 文件等在游戏中都不再需要，游戏中只需要导出的文件。

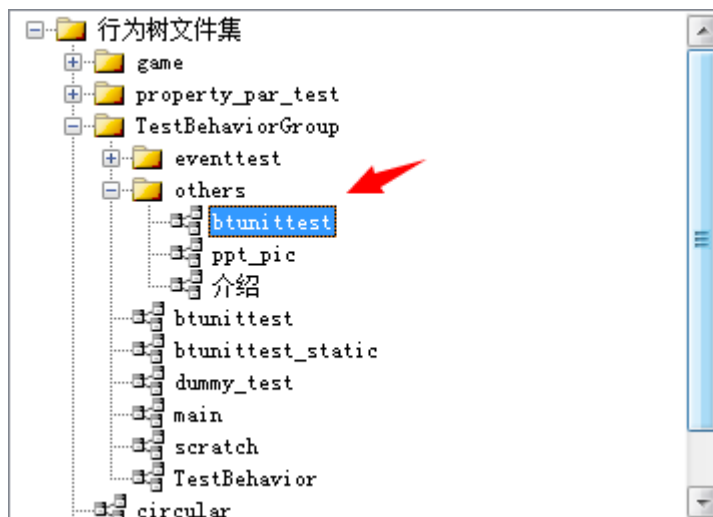
### 5.2 导出路径

下图是 workspace 的设置，可以配置导出路径。所有的 BT 都将导出到这里指定的路径。而游戏中加载（load）BT 的时候，需要调用 BehaviorTree::SetWorkspaceSettings 来指定“导出路径”。游戏中通过 Agent::btload 和 BehaviorTree::Load 来加载 BT 所指定的 BT 的路径也是相对这里的“导出路径”的。

<https://github.com/TencentOpen/behaviac>

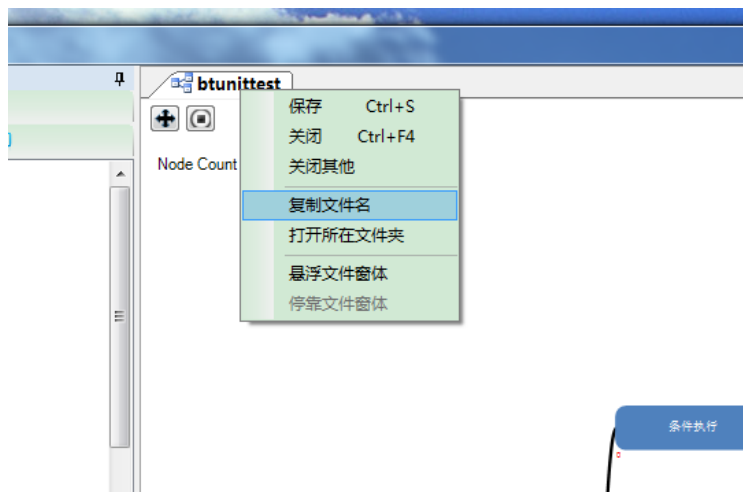


BT 树源文件的 xml 文件导出 xml 或 bson 文件后,在目录中的结构不会被改变。如图, btunittest 在目录结构 TestBehaviorGroup\others 中, 作为源文件是 '源位置'\TestBehaviorGroup\others\btunittest.xml, 而导出后位于 '导出位置' \TestBehaviorGroup\others\



需要注意的是,游戏中通过 BehaviorTree::SetWorkspaceSettings 指定导出路径的时候,该路径是绝对路径或相对于游戏 exe 路径的,通过 Agent::btload 和 BehaviorTree::Load 来加载 BT 所指定的 BT 的相对路径是相对“导出路径”的,特别重要的是如果 BT 是在某个目录中的时候,目录分隔符需要是 '/', 不能使用 '\', 也不需要指定后缀,如上图中的 btunittest,指定它的相对路径为 'TestBehaviorGroup/others/btunittest'。也可以在编辑器中通过如下图右键菜单的 '复制文件名' 来获取这个相对路径:

<https://github.com/TencentOpen/behaviac>

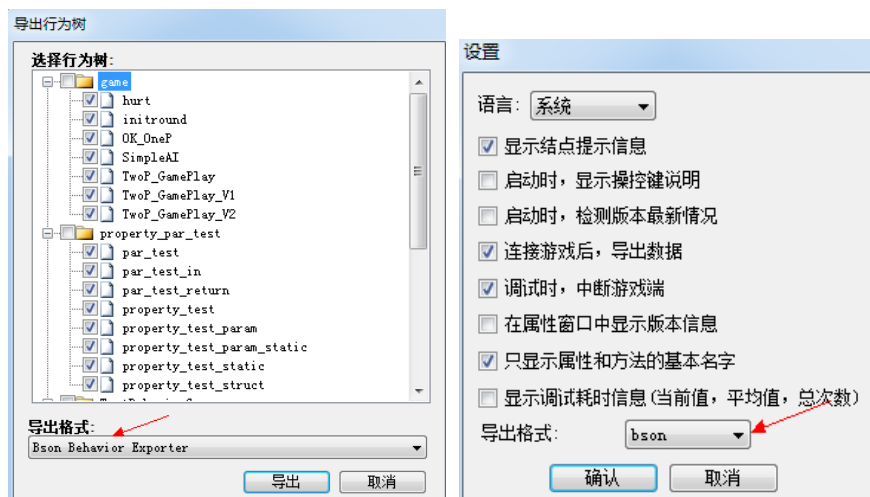


## 5.3 导出格式

目前 behaviac 支持 4 种文件格式 (XML、BSON、C++ 和 C#) 的导出, 详见另一篇文档“Behaviac Tutorial.pdf”中的“导出行为树”。

导出的 BT 格式可以在导出的时候 (“导出行为树”对话框, 左下图) 选择或者在 “设置” 对话框 (右下图) 中指定。“设置” 对话框中指定的导出格式在随后导出 BT 的时候将会被自动选择, 也就是说一般情况下只需要在 “设置” 对话框中指定一次即可。

在项目开发过程中, 建议使用 XML 格式的导出文件, 以便于调试查错等; 而在最终的 release 版本中, 只需导出 C++ 或 C# 格式的行为树文件, 以便提高运行效率。



游戏中调用 BehaviorTree::SetWorkspaceSetttings() 函数来指定导出路径和加载 BT 的格式。

```
/**
 * set the workspace settings
 *
 * 'workspaceRootPath' specifies the file path of the exported path of the workspace file which is configured in the workspace file (.workspace.xml),
 * it can be either an absolute path or relative to the current path.
 * 'format' specify the format to use, xml or bson,
 *
 * the default format is xml.
 *
 * @return false if 'path' is not a valid path holding valid data
 */
static bool SetWorkspaceSettings(const char* workspaceExportPath, EFileFormat format = EFF_xml, float deltaTime = 0.0167f, int deltaFrames = 1);
```

<https://github.com/TencentOpen/behaviac>



游戏中调用 `Agent::btload` 或 `BehaviorTree::Load` 来加载 BT，第一个参数指定某个 BT，相对于导出路径，第二个参数指定是否强制加载，否则如果不是强制加载，则如果已经加载过将直接返回 cache。

```
/**
@param relativePath, relativePath is relative to the workspace exported path. relativePath should not include extension.
the file format(xml/bson) is specified by SetWorkspaceSettings.
@param bForce, the loaded BT is kept in the cache so the subsequent loading will just return it from the cache.
if bForce is true, it will not check the cache and force to load it.

@return
return true if successfully loaded
*/
bool btload(const char* relativePath, bool bForce = false);

/**
relativePath is relative to the workspace path. relativePath should not include extension.
the file format(xml/bson) is specified by SetWorkspaceSettings.

@param bForce
force to load, otherwise it just uses the one in the cache
*/
static bool Load(const char* relativePath, bool bForce = false);
```

## 5.4 命令行导出

BT 除了在编辑器中通过 GUI 来导出，还可以从命令行通过执行命令导出。

```
Usage:
  BehaviacDesigner.exe <workspaceFile> [options]
  options:
    /export=[xml|bson|cpp|cs]
    /help
```

第一个参数 `workspaceFile` 指定 workspace 文件。第二个参数指定导出的格式，如果第二个参数只是 `/export`，则使用 Settings 中指定的导出格式。

例如：

```
c:\Designer\out\BehaviacDesigner.exe .\example_workspace\example.workspace.xml /export=bson
```

# 6 编辑器（designer）

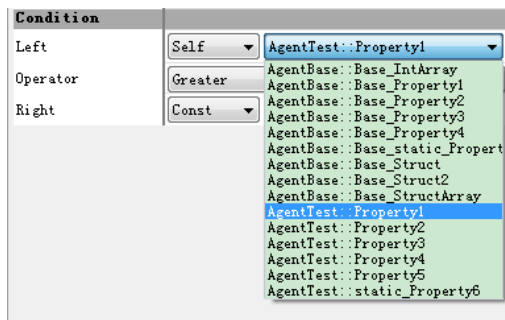
编辑器（designer）用来编辑和调试 BT。编辑器（designer）利用 C++库产生的 meta 信息知道了 Agent 的属性和能力，这样，用户就可以用这个属性和能力来表示 BT 了。和 meta 信息相关的主要是两类节点，条件（Condition）和动作（Action）。

## 6.1 条件（Condition）

Condition 节点对左右参数进行比较，如果结果为 True，返回 success，如果结果为 False，返回 failure，Condition 节点不可能返回 running。通常左参数是 Agent 的某个属性，用户可以

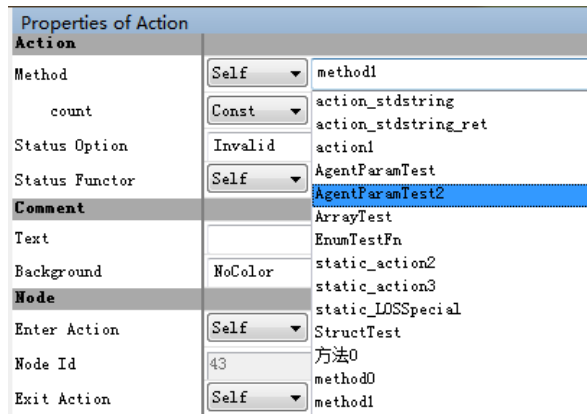
<https://github.com/TencentOpen/behaviac>

从下拉列表里选择属性，右参数是相应类型的常数。



## 6.2 动作（Action）

Action 节点通常对应 Agent 的某个方法（Method），可以从下拉列表里为其选择方法。在设置其方法后，需进一步设置其“决定状态的选项（Status Option）”或“决定状态的函数（Status Functor）”，如下图所示：



有 3 种设置来决定每次 exec 后 Action 节点的状态（success、failure 或 running）：

1. 如果动作节点的方法返回 EBTStatus 值，那么该值就直接作为动作节点的状态，“决定状态的选项”和“决定状态的函数”被禁用无需设置。
2. 否则，需要设置“决定状态的选项”：当选择 Invalid 值时，表明需要进一步设置“决定状态的函数”，否则禁用“决定状态的函数”项，并直接使用“决定状态的选项”所选择的值。
3. 在“决定状态的函数”项中选择的函数，其返回值是 EBTStatus，作为动作节点的状态。该函数只有一个或者没有参数，当动作节点的方法无返回值时，该函数没有参数，当动作节点的方法有返回值时，该函数唯一参数的类型为动作节点方法的返回值类型。

如上图中，

- 当 ‘method1’ 是 void method1(...)的时候，Status Functor 的原型为：EBTStatus StatusFuncor();
- 当 ‘method1’ 是 Return Type method1(...)的时候，Status Functor 的原型为：EBTStatus StatusFuncor(ReturnType param);

## 6.3 注册 Method

C++代码通过 REGISTER\_METHOD 来注册 method，注册过的 method 在动作（Action）或条件

<https://github.com/TencentOpen/behaviac>

(Condition)等节点中可以选择。REGISTER\_METHOD 带有一个参数,表示需要注册的 method, 如下所示:

```
REGISTER_METHOD(action0);  
REGISTER_METHOD(action1);  
REGISTER_METHOD(FnIntReturn);
```

另外, 前面提及的动作节点中“决定状态的函数 (Status Functor)”也需要选择这些注册的函数, 但这些“决定状态的函数”的原型大致需要为如下:

```
behaviac::EBTStatus CheckResult(ReturnType r)
```

即根据返回值来决定 Success, Failure 还是 Running。

下面所示都是可能的形式:

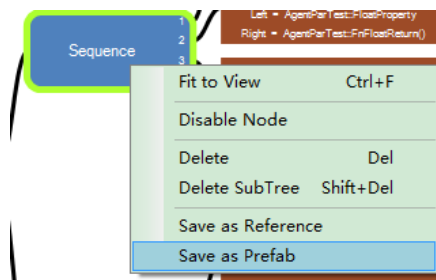
```
EBTStatus CheckResultVoid()  
{  
    EBTStatus s;  
    //..., do what ever to calculate s  
    return s;  
}  
EBTStatus CheckResultInt(int32_t p)  
{  
    EBTStatus s;  
    //..., do what ever to check p to calculate s  
    return EBTStatus;  
}  
EBTStatus CheckResultUInt(uint32_t p)  
{  
    EBTStatus s;  
    //..., do what ever to check p to calculate s  
    return s;  
}  
EBTStatus CheckResultFloat(float p)  
{  
    EBTStatus s;  
    //..., do what ever to check p to calculate s  
    return s;  
}
```

REGISTER\_METHOD 注册的函数如果没有返回值, 则只会在 Action 节点中被选用, 在 Assignment (赋值) 和 Condition (条件) 节点中不会出现。但 REGISTER\_METHOD 注册的函数如果有返回值的时候, 在 Designer 中的 Assignment (赋值) 和 Condition (条件) 节点中作为 ‘Getter’ 可以选择该函数作为右参数或者作为比较的参数, 也可以在 Action 节点中作为 ‘动作’ 被选用, 请使用时注意。

## 6.4 子树

一个 BT 可以作为一个子树通过 ReferencedBehavior 节点添加到其他 BT。



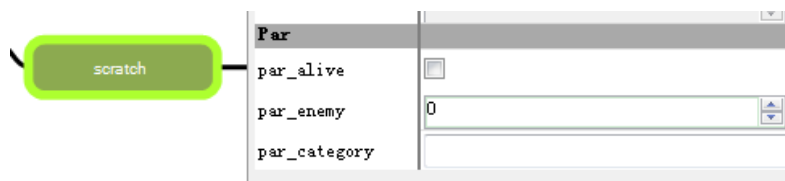


## 6.6 par

par 是 behaviac 中一个重要的部分。par 是 parameter 的缩写，这里作为一个专有‘名词’来特别指代参数的使用。

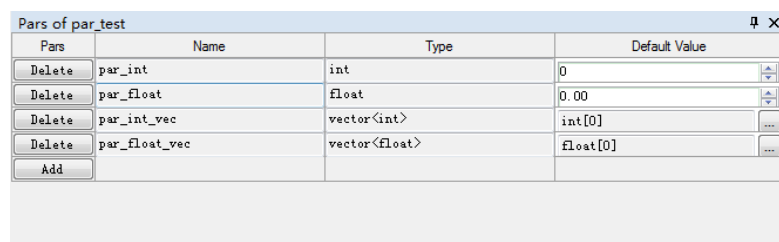
behaviac 中对 Agent 方法（Method）的使用不局限于不带参数的函数，带参数的函数以及有返回值的函数同样支持。

任何一个 BT 都可以定义 pars，所谓 par 是指用户定义的变量，par 可以用作临时中间变量，常数宏，和 C++ 交换数据等。当一个 BT 作为子树被其他 BT 使用时，该子树定义的 pars 可以被赋予不同的初值。

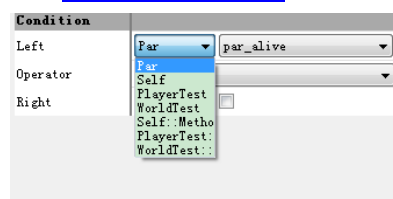


par 的功能使得 behaviac 具有了一定的动态的能力。

需要说明的是，par 非常类似 Blackboard，但 par 的优势是所有有关 par 的使用都是通过 GUI 进行的，用户必须首先创建某个指定类型的 par，然后在可以被使用的地方可以从下拉列表里根据需要的类型选择。



如在 [条件（Condition）](#) 一节中用户可以选择属性（property），其实用户也可以选择‘Par’。



par 也可以作为某个方法（Method）的参数：

<https://github.com/TencentOpen/behaviac>



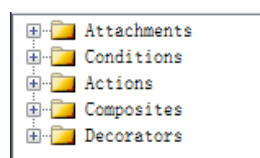
另外在 Assignment 节点里 par 也可以被直接来赋值：



需要指出的是，在 BT 里声明的 par 在执行该 BT 的时候，该 par 才会在该 Agent 里创建。如果是想给该 par 在代码里设初值，只需要直接调用 `SetVariable` 来给该 par 设值即可。在执行 BT 之前或者 `SetVariable` 之前调用 `GetVariable` 试图检查该 par 是否存在是‘非法’的，在 debug 版本下可能会有 assert。

## 6.7 主要节点类型

主要有下列 5 类节点类型：

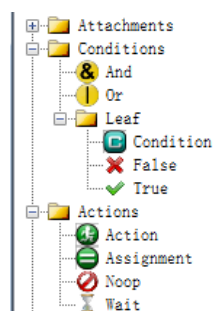


### 6.7.1 附件

附件（Attachments）类型可以附加到相应的节点。特别是 event 类型，需要通过拖拽 BT 到节点来创建。请参考 [Event 作为附件](#)。

### 6.7.2 叶子节点

条件（Conditions）是表述条件的节点类型，动作（Actions）是表示动作的节点类型。这两类节点一般都是作为叶子节点负责具体的逻辑和行为。

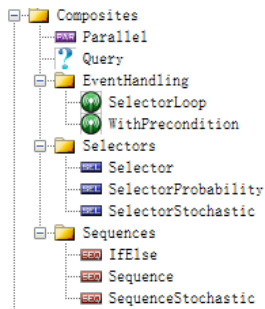


### 6.7.3 分支节点

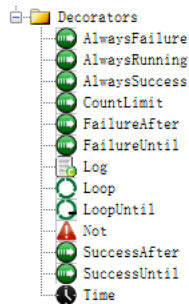
组合（Composites）类型主要是指 Sequence 和 Selector 或类似的控制节点。组合（Composites）

<https://github.com/TencentOpen/behaviac>

类型一般作为 BT 的分支（branch）节点来管理和控制多个子节点。

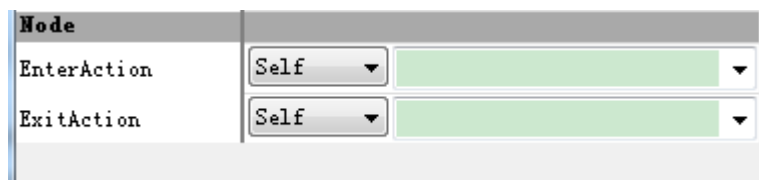


装饰器（Decorator）类型的节点也是作为 BT 的分支（branch）节点，但与组合（Composites）类型不同的是它只管理和控制一个子节点，对该子节点的返回值进行控制。



## 6.8 给节点配置 EnterAction/ExitAction

任何一个节点都可以配置 EnterAction 或 ExitAction，这两个可以为空不予指定。当“进入”节点的时候，这里配置的 EnterAction 就被调用，而当“退出”该节点的时候，这里配置的 ExitAction 就被调用。



EnterAction和ExitAction是进行一些初始化或进行清理工作的好机会。

特别的当ExitAction被调用的时候，可以调用`behaviac::GetNodeExitStatus()`来获得该节点退出的状态是“成功”还是“失败”，从而做些相应的清理任务。需要指出的是`behaviac::GetNodeExitStatus()`只能在ExitAction里调用，在其他地方被调用的时候将会触发运行时的错误。

## 6.9 配置 Config.xml

在安装目录（BehaviacDesigner.exe 所在目录）有一个 config.xml 的文件，该文件如图所示：

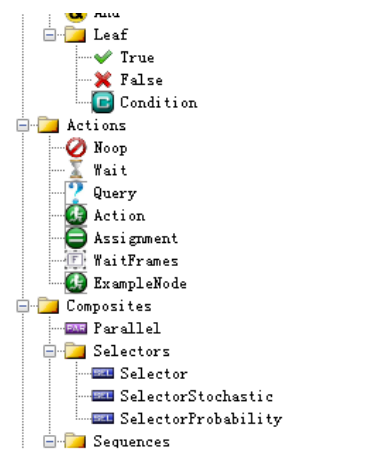
```

<!--
the nodes listed here are hidden(filtered) in the designer's node list.
-->
<Configuration>
  <FilterNodes>
    <node FullName="PluginBehaviac.Nodes.DecoratorTime" />
    <node FullName="PluginBehaviac.Nodes.Wait" />
    <node FullName="PluginBehaviac.Nodes.Query" />
  </FilterNodes>
</Configuration>

```

在这个文件里可以指定那些不想在 Node List 中出现的节点类型。如在上图中指定的三个节点类型就不会出现在 Designer 的节点列表里，从而用户就不可能创建该节点类型的节点（但如果打开的 BT 里已经有了这个节点，则不影响该节点的使用，只是不能添加该类型新的节点了）。

那么从哪里可以知道这个‘FullName’呢？当在设置里选择语言为 English 或英文的时候，



一般情况下，这里列出的就是节点类型的名字，FullName 是这里的名字加上该节点类型所在 namespace 的名字组合而成，如果你想隐藏掉自带的节点类型，则 namespace 都是 PluginBehaviac.Nodes，否则请询问实现相应节点类型的程序员知悉所在的 namespace。

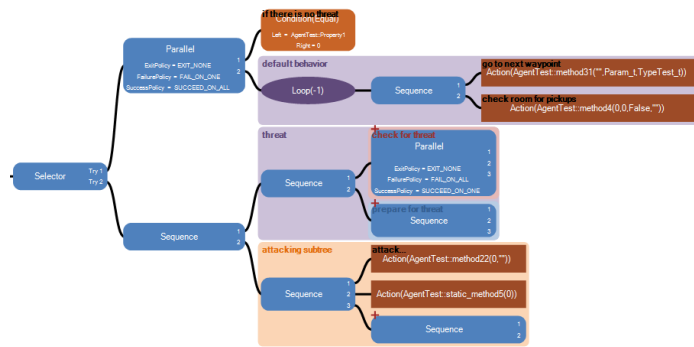
## 7 事件处理

运行 BT 的时候，发生事件的时候怎么响应用于切换行为树是个大问题。

### 7.1 依靠 Parallel 节点

依靠 Parallel 节点处理 event，需要把 event 用 precondition 的形式表达并且需要监控该 precondition，当该 precondition 不满足的时候退出。概念上不清晰，使用上比较繁琐。

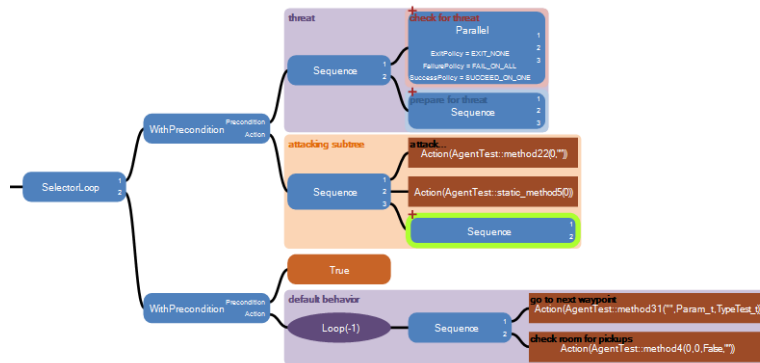




## 7.2 SelectorLoop 和 WithPrecondition

SelectorLoop 和 WithPrecondition 作为对传统 BT 的扩展对于类型事件的处理就自然多了。SelectorLoop 和 WithPrecondition 只能配对使用，即 SelectorLoop 只能添加 WithPrecondition 作为它的子节点，WithPrecondition 也只能作为 SelectorLoop 的子节点被添加。

- SelectorLoop 是一个动态的选择节点，和 Selector 相同的是，它选择第一个 success 的节点，但不同的是，它不是只选择一次，而是每次执行的时候都对其子节点进行选择
- WithPrecondition 有 precondition 子树和 action 子树。只有 precondition 子树返回 success 的时候，action 子树才能够被执行。



## 7.3 Event 作为附件

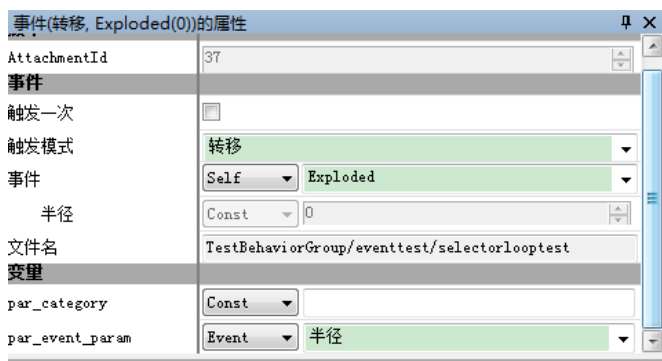
任何一个 BT 都可以作为事件子树，作为 event 附加到任何一个节点上(用鼠标拖动 BT 到节点)。当 Runtime 运行该 BT 的时候，如果发生了某个事件，可以通过 Agent::FireEvent 来触发该事件，则处于 running 状态的节点，**从下到上**都有机会检查是否需要响应该事件，如果有该事件配置，则相应的事件子树就会被触发。



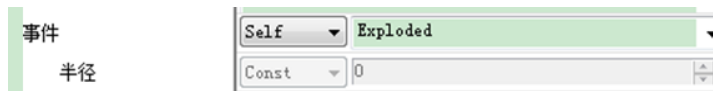
如同用 REGISTER\_METHOD 来注册函数供动作（Action）节点使用，需要用 REGISTER\_EVENT 来注册事件（Event），从而如上图才可以从下拉列表中选择。

```
REGISTER_EVENT("EventDead").SETSTATIC();
REGISTER_EVENT("Exploded", int).PARAM_DISPLAY_INFO(L"半径", L"半径参数的说明");
REGISTER_EVENT("HurtBy", int, bool).PARAM_DISPLAY_INFO(L"武器").PARAM_DISPLAY_INFO(L"reload");
REGISTER_EVENT("Killed", int, bool, float).PARAM_DISPLAY_INFO(L"weapon").PARAM_DISPLAY_INFO(L"同盟");
```

注册的事件可以有参数，如上图“Exploded”就有一个参数作为 Exploded 发生时候的半径。该参数在该事件被触发（Agent::FireEvent）的时候作为 Agent::FireEvent 的参数被指定，如下图在配置该事件的时候，可以选择把该事件的参数传给某个 par，在响应该事件相应子树的时候，就可以使用该 par 来使用该事件的参数了。



例如上图中：事件“Exploded”就有一个参数作为 Exploded 发生时候的半径，



响应该事件的子树是 selectorlooptest，该子树有一个 par，“par\_event\_param”，



可以在下拉列表里选择“半径”赋值给该 par。那么当“Exploded”被触发从而 selectorlooptest 被更新的时候，“par\_event\_param”就是“Exploded”的参数“半径”的值。

事件还有一个属性 **触发模式** 可以配置。“触发模式”控制该事件触发后对当前 BT 的影响以及被触发的子树结束时如何恢复，有转移（Transfer）和返回（Return）两个选项。

- **转移(Transfer):** 当前的 BT 被中断 (abort) 和重置 (reset), 新的 BT 被设置为当前 BT, 当这个新的 BT 结束的时候。
- **返回(Return):** 子树结束时返回控制到之前打断的地方继续执行。当前的 BT 直接‘压’到执行堆栈上而不被 abort 和 reset, 新的 BT 被设置为当前 BT, 当这个新的 BT 结

<https://github.com/TencentOpen/behaviac>

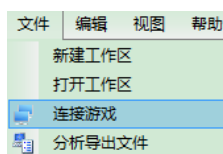
束的时候，原本的那个 BT 从执行堆栈上‘弹出’从当初的节点恢复执行。

## 8 调试

游戏运行时，编辑器可以实时的连接到游戏对 BT 的运行情况进行检查和调试。

### 8.1 连接游戏

可以通过菜单

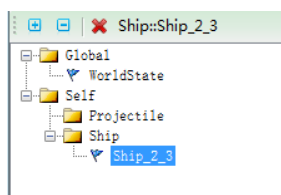


或快捷按钮连接游戏。

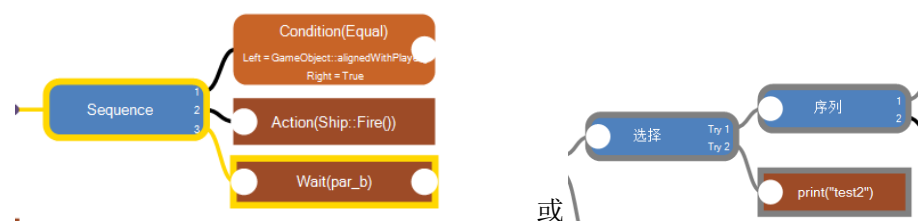


### 8.2 检查 BT 的运行情况

和游戏建立连接后，可以选择感兴趣的 Agent 检查它当前活动的 BT。





如图将以图形的形式显示 BT 的运行情况：



### 8.3 边框颜色

BT 的运行情况以不同的**边框颜色**来显示不同的状态。

- 黄色  代表 running 状态。
- 灰色  代表在该帧该节点有运行更新过，但在该帧内就结束了，即经历了从

<https://github.com/TencentOpen/behaviac>

enter, running, exit。

- 没有边框  代表在该帧该节点没有运行更新过。

## 8.4 Agent::SetIdMask 和 Agent::SetIdFlag

缺省情况下 Agent 不被 Debug，只有 Agent::IsMasked() 返回 True 的 Agent 才被 Debug。你需要调用 Agent::SetIdMask 和 Agent::SetIdFlag 打开对相应 Agent 类型的 Debug。

如果 Designer 连上游戏后看不到你的 Instance 或者什么 Instance 都没有，很可能就是没有调用 Agent::SetIdMask 和 Agent::SetIdFlag。

```
bool Agent::IsMasked() const
{
    return (this->m_idFlag & Agent::ms_idMask) != 0;
}
```

例如当 ms\_idMask = 0xFFFFFFFF 的时候，你可以设置你飞船的 m\_idFlag 为 1，炮弹的 m\_idFlag 为 0，那么你将可以调试你的飞船，但炮弹将不被调试。

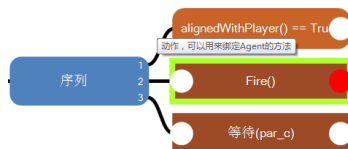
## 8.5 查看或修改属性（property）

Properties of Ship:Ship_2_3		
Name	Type	Value
GameObject::age	long	203168
GameObject::HP	uint	0

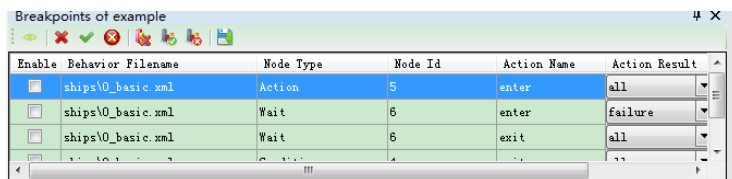
pars 的值也可以查看和修改。

## 8.6 设置断点

在节点的左右两侧，双击鼠标左键，红色的圆圈显示设置了断点，白色的圆圈显示该断点 Disabled。左侧和右侧分别表示 Enter 和 Exit。



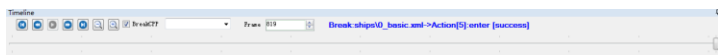
可以选择是否要断下 C++端 ☒ BreakC++。



Enable	Behavior	Filename	Node Type	Node Id	Action Name	Action Result
<input checked="" type="checkbox"/>		ships\0_basic.xml	Action	5	enter	all
<input type="checkbox"/>		ships\0_basic.xml	Wait	6	enter	failure
<input type="checkbox"/>		ships\0_basic.xml	Wait	6	exit	all

## 8.7 时间轴（Timeline）

可以选择具体某一帧，然后检查该帧时 BT 的运行情况，属性或 par 的值。



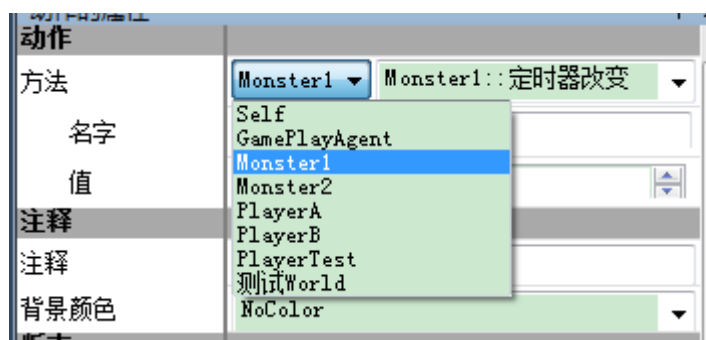
## 9 注册名字和实例

如图，Agent::RegisterName 可以注册一个名字，该注册的名字表示游戏中某个 Agent 类的实例：

```
Agent::RegisterName<PlayerTest>("PlayerA");
Agent::RegisterName<PlayerTest>("PlayerB");

Agent::RegisterName<PlayerTest>();
Agent::RegisterName<WorldTest>();
```

在 Designer 中当需要选择函数（Method）或属性（Property）的时候，在左侧的 combo 里该名字在下拉列表里供选择，从而可以选择该名字对应 Agent 类的函数（Method）或属性（Property）。这样子选择后的效果就是在游戏中执行该 BT 的时候，将会访问（access）该名字所绑定的实例对应的函数（Method）或属性（Property）。



需要说明的是，游戏中需要调用 Agent::CreateInstance 或 Agent::BindInstance 创建实例并且绑定到该名字。如果指定的名字没有注册或注册的类型不同或该名字已经绑定，则调用 Agent::CreateInstance 的时候会有 runtime 的错误。

当调用 Agent::RegisterName 和 Agent::CreateInstance 而没有指定名字的时候，该 Agent 类型的名字将作为名字被注册和绑定。

```
Agent::CreateInstance<PlayerTest>("PlayerA");  
Agent::CreateInstance<PlayerTest>("PlayerB");  
  
Agent::CreateInstance<PlayerTest>();  
Agent::CreateInstance<WorldTest>();
```

```
Agent::BindInstance("PlayerA", pPlayerA);
```

当游戏中有一个 TPlayer 的时候，就可以如下图注册和创建并且绑定该 TPlayer 的实例。

```
Agent::RegisterName<TPlayer>();  
Agent::CreateInstance<TPlayer>();
```

但当该 TPlayer 被敌人打死后又 spawn 了一个新的实例的时候，可以首先调用 Agent::UnbindInstance 解除绑定，之后再创建或通过 Agent::BindInstance 重新绑定一个新的实例。

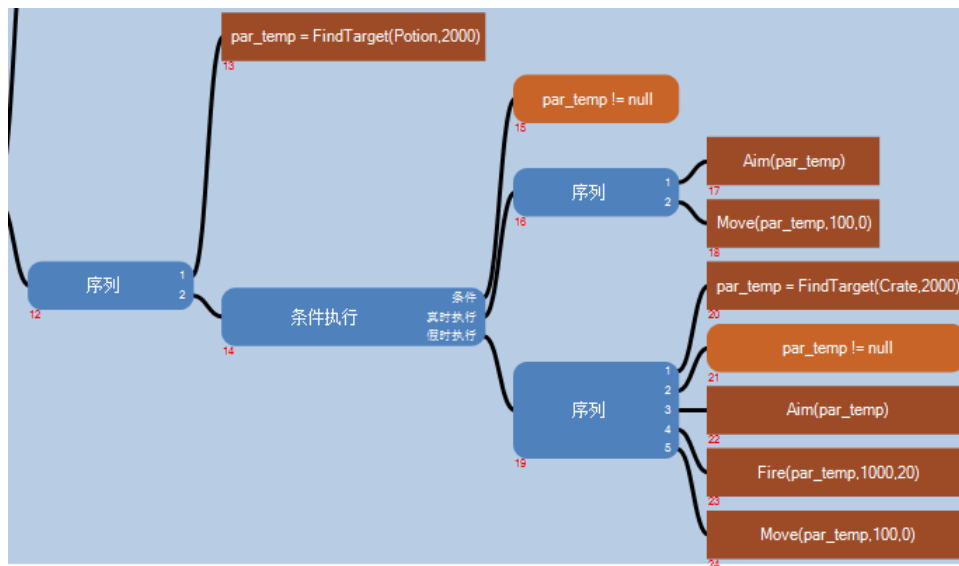
注册名字和绑定实例的**设计意图**主要是为了支持 singleton 或者类似确定的全局性的实例(同一个类可能会有若干个实例而不是仅仅有一个实例)。例如：player，camera，director 等。通过名字就可以在 Designer 中访问相应实例的 Method 或 Property。

## 9.1 关于注册和绑定的说明

只有通过 RegisterName 注册过的‘名字’才在导出元信息 meta 文件的时候被导出。而 BindInstance 和 CreateInstance 等和 Instance 相关的函数在需要执行 BT 的时候才需要，如果只是导出元信息则**不需要**。

# 10 对 Agent 类型的指针的访问

有的时候，某些函数需要访问 Agent 类型的指针来根据该 Agent 来实现某些功能。例如行为中需要首先找到 target，然后朝向该 target 并且向该 target 射击和移动。如下图：



对应的 C++函数则如下图（请注意第一个参数）：

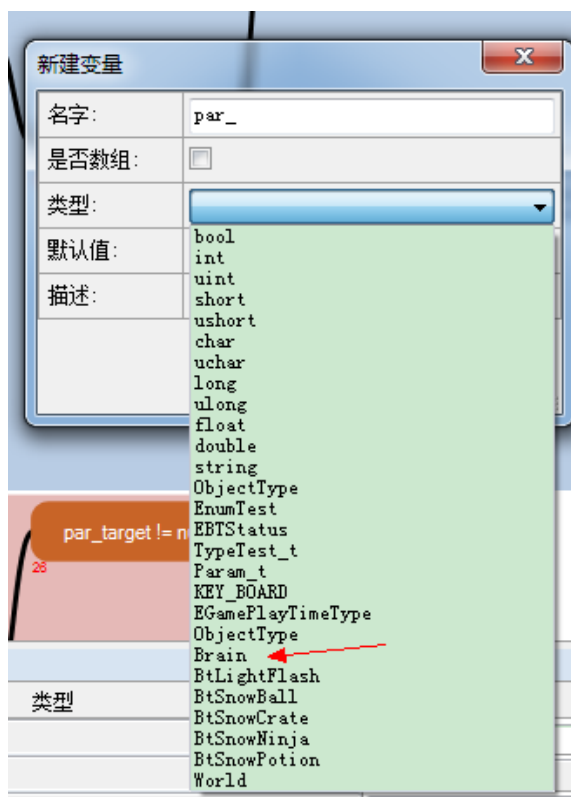
```
void Fire(Brain* target, float fireRange = 2500.0f, float fire_scale = 1.0f);

void Aim(Brain* target);
void Move(Brain* target, float farRange = 3000.0f, float closeRange = 600.0f);
```

par\_temp 是一个 par:



在创建 par 的时候，在类型的下拉列表里可以选择 Agent 类型作为该 par 的类型：



这个功能的设计意图在于实现 coordination 或动态的处理等。就如同在上面的例子中通过 <https://github.com/TencentOpen/behaviac>

FindTarget 来动态的确定某个 target，然后，后续的行为中可以对该 target 做出处理。

## 11 加载

```
/**
 * relativePath is relative to the workspace exported path. relativePath should not include extension.
 * the file format(xml/bson) is specified by SetWorkspaceSettings.
 *
 * @param bForce
 * force to load, otherwise it just uses the one in the cache
 */
static bool Load(const char* relativePath, bool bForce = false);
```

Workspace::Load(Agent::btload 实际上直接调用 Workspace::Load)负责加载 (Load) BT 树。上图是 Workspace::Load 的原型。加载的 BT 树被添加到 Cache 中，后续通过 Workspace::Load 加载的时候，如果 ‘bForce’ 为 true，则强制读取文件重新加载，否则直接从该 Cache 中返回。

Workspace::Load 加载 BT 的时候，根据设定的格式选择使用 xml、bson 还是 cpp、c#等格式。

### 11.1 热加载

我们知道，在游戏运行的时候，Workspace::SetWorkspaceSettings 被调用来指定 workspace 的导出路径：

```
Workspace::SetWorkspaceSettings("../integration/unity/Assets/Resources/example_workspace/exported", format);
```

在开发版中，当热加载功能激活时（可以在 Workspace::SetWorkspaceSettings 中的第 3 个参数关闭），该导出路径会被“监听”，即当有 BT 文件（仅 Xml 或 Bson 格式）被重新导出的时候，会被自动的重新加载，相应 Agent 实例中的 BT 树也被重新创建。如果指定的格式是 C++或 C#的时候不支持热加载。

### 11.2 更新包

当游戏已经发布后，可能会有更新包更新游戏，BT 可能被更新：

- 如果更新包可以包含可执行程序的话，更新的 BT 可以通过更新的 C++文件或 C#文件从而 build 到更新的可执行程序里来更新。
- 但如果这个更新包只能是资源的时候，更新的 BT 就只能是更新的 Xml 或 Bson 文件了。通过 Workspace::SetWorkspaceSettings 的第二个参数 format 设置文件格式为 EFF\_default，则游戏使用更新包的逻辑是检查 BT 的导出路径，如果有更新的 Xml 或 Bson 文件，则使用之，而不再使用原始 build 到 exe 的 BT 或原本的 BT。

## 12 C++程序使用 behaviac

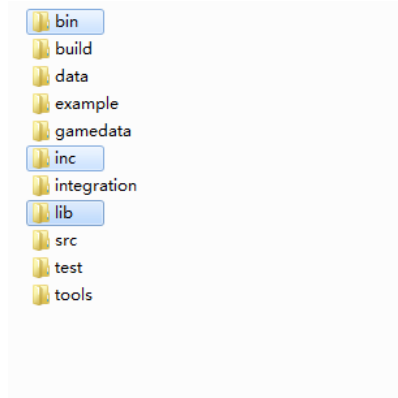
使用 behaviac 库，需要

<https://github.com/TencentOpen/behaviac>



- include 头文件
- 以及 link 库
- 运行时可能需要 dll 文件（如果用的是 dynamic 版本的话）

下图所示的目录分别是 include 的 path，lib，以及 dll 的路径。



上图是安装后的目录结构。

- build 目录中是项目文件（vcproj, sln 等文件），需要说明的是，安装包中，build 目录仅包含了 unittest 的项目文件，若需要完整项目文件请和项目组联系。
- src 目录是 C++ runtime 的源码。
- test 目录是 unittest 的源码
- tools 目录是 Designer 的执行程序等。Designer 的源码在安装包中没有包含，若有需要请和项目组联系。
- integration 目录是 C# runtime 及 unittest 和 unity 测试的源码和项目文件。

## 12.1 版本说明

在 lib 目录下，有如下命名的 libs:

behaviac\_debugdll\_win32\_vs2010  
behaviac\_profiledll\_win32\_vs2010  
behaviac\_releasedll\_win32\_vs2010

behaviac\_debugstatic\_win32\_vs2008  
behaviac\_profilestatic\_win32\_vs2008  
behaviac\_releasestatic\_win32\_vs2008

文件名中有 dll 的是 dll 版本，有 static 的是静态库版本。对于 DLL 版本，在 bin 目录还有相应的 dll。此外根据你的版本的不同还会有 vs2005, vs2013, gmake 等版本。

## 12.2 开发版本和最终版本

宏 `BEHAVIAC_RELEASE` 定义的时候是最终版，`BEHAVIAC_RELEASE` 在 Debug 和 Profiling 版本中没有定义，在 Release 版本中有定义。最终版本中，`behaviac::Agent::ExportMetas` 和

<https://github.com/TencentOpen/behaviac>

behaviac::Socket::SetupConnection 等函数将什么都不做(下述章节中查看相应函数的说明)。当然在某些情况下,比如需要在 Release 版本中也做出某些测试的时候,也可以选择在项目文件中定义 `BEHAVIAC_RELEASE`。

在开发版中,可以通过 behaviac::Config::IsLogging 和 behaviac::Config::IsSocketing 来控制是否要 Log 到文件或是否和编辑器的连接。

## 12.3 主要步骤

```
//the first function to call to start behaviac module
behaviac::Start();

//register agents
behaviac::Agent::Register<TestAgent>();

//after all the 'Agent::Register', export meta info,
//this is only needed to execute when you Agent's structure is modified
behaviac::Workspace::ExportMetas("agents\\testAgents.xml");

//set the workspace export path from which all the BT's path is relative to
behaviac::Workspace::SetWorkspaceSettings("workspace\\exported");

//behaviac::Workspace::SetDeltaFrameTime(0.0167f);
behaviac::Workspace::SetDeltaFrames(1);

//create agent
TestAgent *obj = behaviac::Agent::Create<TestAgent>();
//load BTs, the path is relative to the workspace's export path
obj->btload("test_selector/test_selector_multi_tick");
obj->btsetcurrent("test_selector/test_selector_multi_tick");

//enable to debug it that it can appear in the connected designer
obj->SetIdFlag(1);
TestAgent::SetIdMask(0xffffffff);

//block or not block
bool bBlock = false;
behaviac::Socket::SetupConnection(bBlock);

while (true)
{
    //please use the World::btexec which will exec all the agents instances
    //if you directly call obj->btexec(interval), the debugging can't work properly that
    //the conected designer can't display the updated status, etc.
    //obj->btexec();
    behaviac::World::GetInstance()->btexec();

    Sleep(1000);
}

//clean up
BEHAVIAC_DELETE(obj);
TestAgent::Agent::UnRegister<TestAgent>();
behaviac::Stop();
```

## 12.4 几点特别注意

- `behaviac::Agent::ExportMetas` 是用来导出元信息的。元信息是仅供 Designer 使用的，Runtime 不需要使用元信息，元信息只需要开发版本中当 Agent 有改变的时候导出一次就可以，不需要每次都运行该函数。发布版本中不需要运行该函数。然而在开发版本中，也可以每次都运行该函数，这样子可以确保任何 Agent 的改动都可以更新元信息 meta 文件，只要 Agent 没有改变，导出的元信息 meta 文件也将是一样的。需要指出的是 `behaviac::Agent::ExportMeta` 仅在开发版本中有定义，在非开发版本中的定义是空函数。
- `behaviac::BehaviorTree::SetWorkspaceSettings` 指定导出路径，后续 load 行为树的时候，都是相对于这个路径的。
- `SetIdFlag` 和 `SetIdMask` 对于连接游戏和 Designer 调试查看 BT 的运行情况至关重要。缺省情况下 Agent 是不被调试的，只有通过恰当的设置这两个值才能控制打开相应的 Agent 的调试。
- 如果需要连接游戏和 Designer 调试查看 BT 的运行情况，则需要调用 `behaviac::Socket::SetupConnection`，该函数的第一个参数表明是否要阻塞等待 Designer 连接。`behaviac::Socket::SetupConnection` 仅在开发版本中有定义，在非开发版本中的定义是空函数。
- 如果需要连接游戏和 Designer 调试查看 BT 的运行情况，则需要调用 `behaviac::World::GetInstance()->btexec` 来执行各个 agent，如果不通过 `behaviac::World::GetInstance()->btexec` 来执行，只是直接调用 agent 的 `btexec` 的话，可能不能正确连接 Designer 或连接的 Designer 不能调试查看 BT 的运行情况。
- `Behaviac::Workspace` 类可以指定 `DeltaFrames` 或 `DeltaFrameTime` 来表明两次调用之间的时间间隔或帧数间隔。
- 需要指出的是这个间隔大部分节点根本没有用到。目前只有 `Wait`，`DecoratorTime`（`WaitFrames`，`DecoratorFrames`）用到了。而且 `Wait`，`DecoratorTime`（`WaitFrames`，`DecoratorFrames`）实际是两组，`Wait`，`DecoratorTime` 用的是 `DeltaFrameTime`，而 `WaitFrames`，`DecoratorFrames` 用的是 `DeltaFrames`。[配置 Config.xml](#) 中可以用来配置需要隐藏掉不用的那组。缺省的，`Wait`，`DecoratorTime` 是被隐藏掉的，也就是说缺省情况下，用的是 `WaitFrames`，`DecoratorFrames`。具体细节可以参考 `Wait.cpp`，`DecoratorTime.cpp`，`WaitFrames.cpp`，`DecoratorFrames.cpp` 等。

## 12.5 执行 BT，World::btexec

```
EBTStatus World::btexec()
{
    behaviac::Profiler::GetInstance()->BeginFrame();

    Workspace::LogFrames();
    Workspace::HandleRequests();

    if (Workspace::GetAutoHotReload())
    {
        Workspace::HotReload();
    }

    EBTStatus s = super::btexec();
    BEHAVIAC_UNUSED_VAR(s);

    if (this->m_bTickAgents)
    {
        this->btexec_agents();
    }

    behaviac::Profiler::GetInstance()->EndFrame();

    //TODO: BT_INVALID
    return BT_RUNNING;
}
```

World::btexec除了调用super::btexec以及btexec\_agents来更新自己和它所包含的其他Agents上的BT外，还做了诸如LogFrames和HandleRequests等其他事情，这些事情是为了实现和Designer的交互调试的必要支持。

如下图所示World::btexec\_agents，执行所有agents的BT。每个agent都有一个priority，根据该priority，执行agents的顺序不同，priority越高越早被执行。

```
void World::btexec_agents()
{
    std::make_heap(this->m_agents.begin(), this->m_agents.end(), HeapCompare_t());

    for (PriorityAgents_t::iterator it = this->m_agents.begin(); it != this->m_agents.end(); ++it)
    {
        HeapItem_t& pa = *it;

        for (Agents_t::iterator ita = pa.agents.begin(); ita != pa.agents.end(); ++ita)
        {
            Agent* pA = ita->second;

            if (pA->IsActive())
            {
                pA->btexec();

                //in case m_bTickAgents was set to false by pA's bt
                if (!this->m_bTickAgents)
                {
                    break;
                }
            }
        }
    }

    if (Agent::IdMask() != 0)
    {
        int contextId = this->GetContextId();
        Context& c = Context::GetContext(contextId);

        c.LogStaticVariables(0);
    }
}
```

这样子的一个缺陷是每个agent的执行频率都是一样的。

实际上，调用World::btexec只是更新agent执行BT的“快捷方式”而已，在游戏中完全可以用自己定制的执行逻辑。

**请注意**，在自己定制执行的时候，除了要调用btexec来执行BT，也还需要调用LogFrames

<https://github.com/TencentOpen/behaviac>

来通知编辑器新的帧数和 `HandleRequests` 来处理从编辑器发过来的消息，否则就不能正常的调试，但这两个函数仅在需要连接编辑器调试的时候需要，而且每帧只需要调用一次。

## 12.6 FileManager

不同的游戏对资源的管理组织方式可能会不同，有的可能把资源按照某种方式打包压缩等处理。如果游戏有自己对资源的组织方式，则需要重载 `CFileManager` 提供相应的函数实现，特别是：

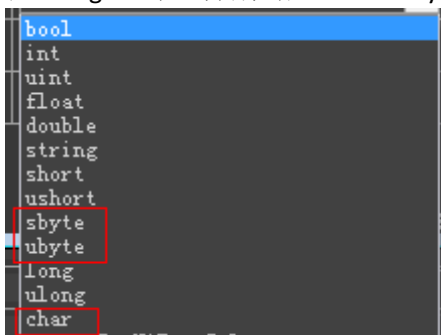
```
virtual IFile* FileOpen(const char* fileName, CFileSystem::EOpenAccess iOpenAccess = CFileSystem::EOpenAccess_Read);  
virtual void FileClose(IFile* file);  
virtual bool FileExists(const char* fileName);
```

`CFileManager` 内部被当作一个 singleton 来使用，当在游戏的初始化阶段创建了自己的重载类，则该实例被使用，否则，系统的缺省实现被使用。

## 12.7 关于 char, signed char, unsigned char 等的说明

此外，需要对 char, signed char, unsigned char 做出一些必要的说明：

1. 在 c++中，char, signed char, unsigned char 是三个不同的类型，特别的 char 和 signed char 是不同的类型
2. 在 Designer 中，分别对应 char, sbyte, ubyte



3. 对于 C#，char, sbyte, byte 是基本类型，分别对应 Designer 中的 char, sbyte 和 ubyte

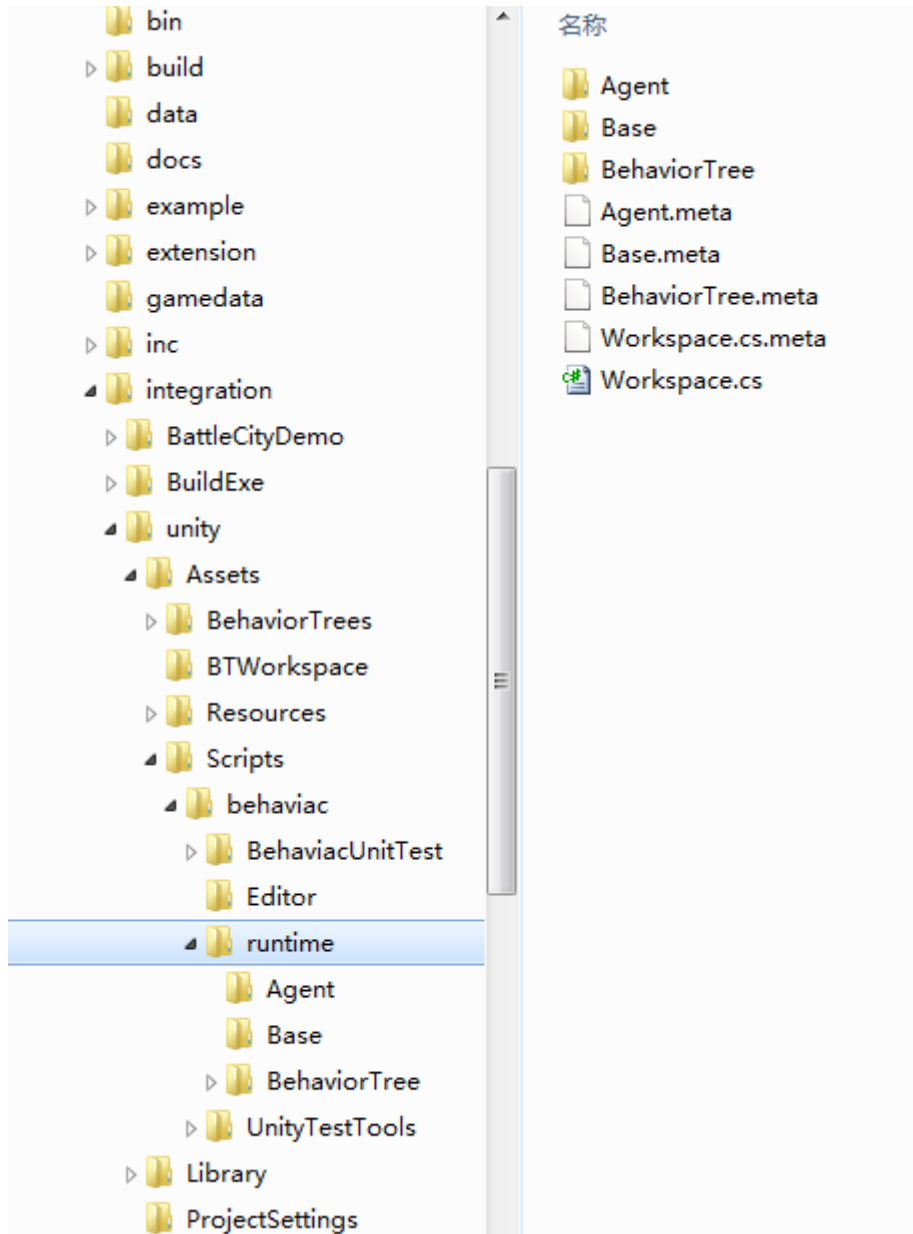
## 12.8 关于整合 behaviac 库的 FAQ

在 Linux 平台整合 behaviac 库时，需要注意：

1. 编译参数需设置 `-fno-rtti`，否则编译会提示找不到 behaviac 库中类的 typeinfo。
2. 如果 behaviac 库编译为 debug 版本，那么自己的项目也需要编译为 debug 版本，并且需在编译参数中添加 `_DEBUG` 宏。

## 13 Unity 中使用 behaviac

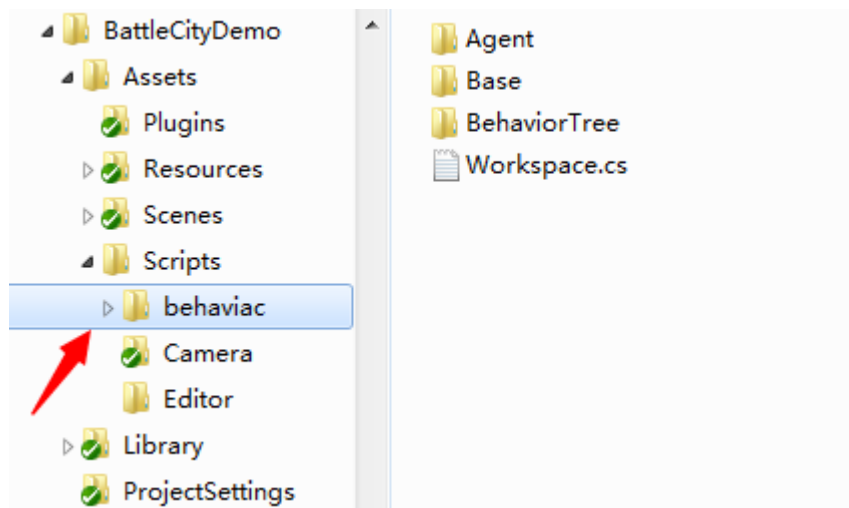
如下图所示，在安装目录里有 integration 的目录，该目录里包含有 C# Runtime 源码。  
在 C# 及 Unity 中使用 behaviac 与在 C++ 中基本一致，所使用的类名、函数名和 C++ 中尽量保持一致。



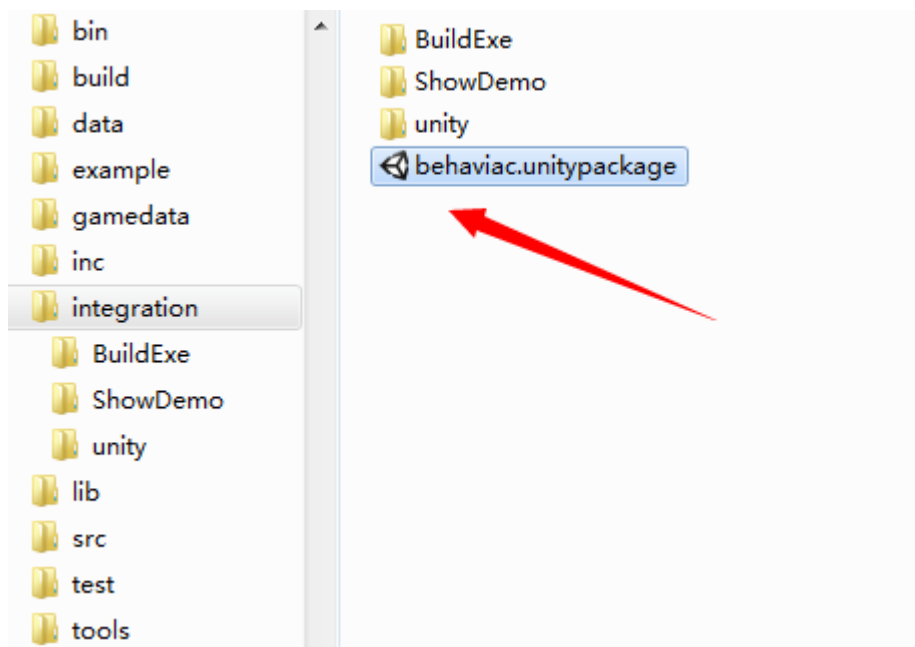
### 13.1 使用 behaviac 需要的文件

把目录\integration\unity\Assets\Scripts\behaviac\runtime 拷贝到自己的 Unity 项目目录下的 Assets 里就可以使用 behaviac 了,注意可以不必要 copy 里面的 meta 文件。如图，假若 BattleCityDemo 是你的项目，

<https://github.com/TencentOpen/behaviac>



或者在 unity 导入 behaviac.unitypackage 到项目也可：



## 13.2 版本说明

当定义 `BEHAVIAC_RELEASE` 的时候，所有调试功能，开发功能都没有实现。甚至 `ExportMetas` 这个函数也是空的。

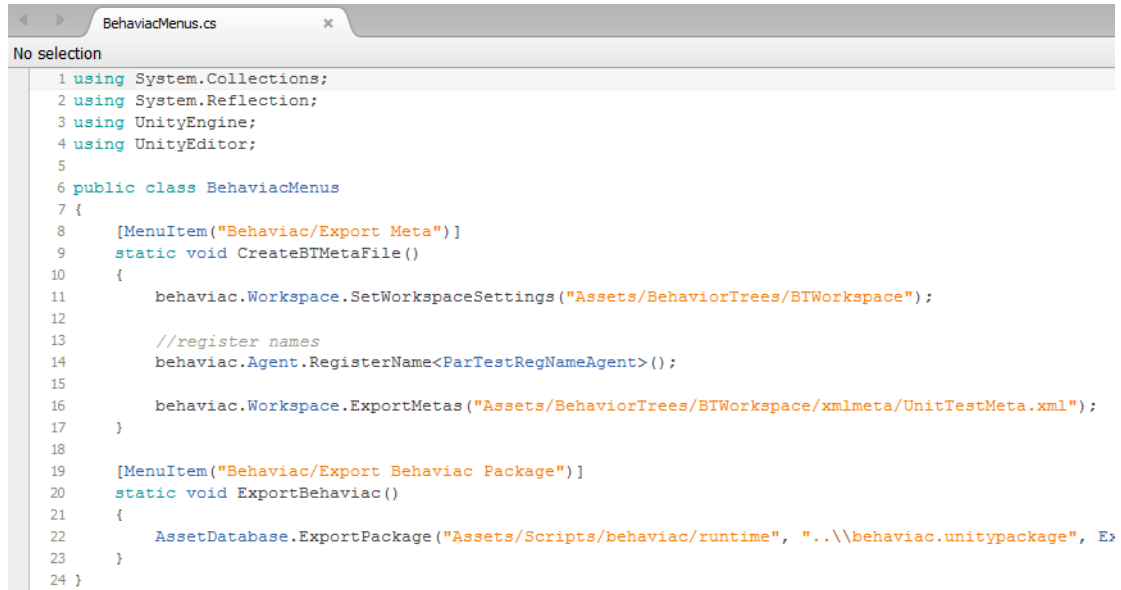
所以当发布版本或是做手机版本的时候，请定义 `BEHAVIAC_RELEASE` 以获得最优的性能和最少的内存使用。

但是需要指出的是当定义 `BEHAVIAC_RELEASE` 的时候：

- 由于 `ExportMetas` 是空的，即便调用了 `ExportMetas`，meta 数据也是不会被更新的
- 此外，调试功能也是没有实现的，运行中的游戏不能和编辑器建立连接，也没有 log 输出等。

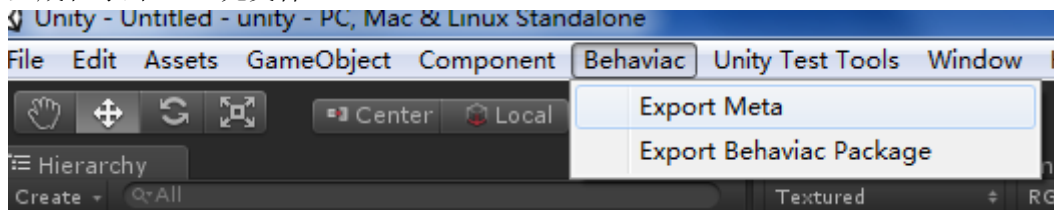
## 13.3 Editor Script

此外，unity\Assets\Scripts\behaviac\Editor 下有 BehaviacMenus.cs:



```
1 using System.Collections;
2 using System.Reflection;
3 using UnityEngine;
4 using UnityEditor;
5
6 public class BehaviacMenus
7 {
8     [MenuItem("Behaviac/Export Meta")]
9     static void CreateBTMetaFile()
10     {
11         behaviac.Workspace.SetWorkspaceSettings("Assets/BehaviorTrees/BTWorkspace");
12
13         //register names
14         behaviac.Agent.RegisterName<ParTestRegNameAgent>();
15
16         behaviac.Workspace.ExportMetas("Assets/BehaviorTrees/BTWorkspace/xmlmeta/UnitTestMeta.xml");
17     }
18
19     [MenuItem("Behaviac/Export Behaviac Package")]
20     static void ExportBehaviac()
21     {
22         AssetDatabase.ExportPackage("Assets/Scripts/behaviac/runtime", "..\\behaviac.unitypackage", ExportPackageOptions.ReplaceExistingFiles);
23     }
24 }
```

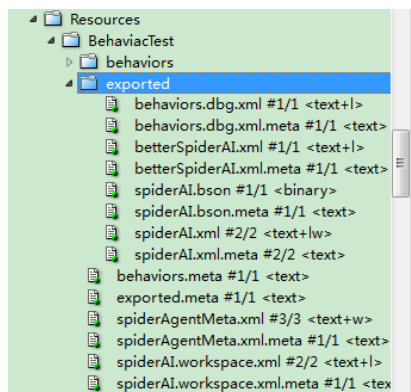
可以在项目中类似的添加 Editor script，从而可以在 Unity Editor 里通过 Menu 的方式手工的生成和导出 meta 元文件。



但是请注意，通过此方式手工的生成和导出 meta 元文件将有可能导致当类信息更新而没有及时更新 meta 元文件的**风险**！

## 13.4 Resources

基于 unity 开发对 Resources 加载的要求，导出的 BT 需要放在 Resources 目录下，如图：



‘exported’ 目录是 workspace 的导出路径，必须放在 Resources 目录下，而 workspace 的其

<https://github.com/TencentOpen/behaviac>



他目录和文件在运行时是不需要的，则不需要放在 Resources 目录下，虽然放在 Resources 目录下也不影响运行。

另外，由于导出的格式可以是 xml，也可以是 bson(此格式的时候，文件扩展名是.bson.bytes)，而运行的时候一般要么用 xml，要么用 bson，所以测导出路径中也仅需要么导出 xml，要么导出 bson，不需要两种格式都导出。请参考[导出格式](#)和[更新包](#)。

### 13.4.1 FileMansager

如果有自己对资源的打包方式或管理方式，需要重载 FileManager。

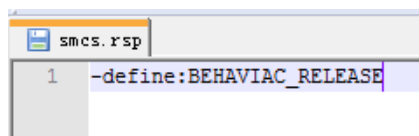
## 13.5 Logging 和 Connect（连接 Designer）

缺省情况下，beHAVIAC 的 C# script 是开发模式，特别在 WindowsEditor 和 WindowsPlayer 下，Logging 和 Conenect 是缺省打开的。但在其他非桌面平台（android，ios 等）下，Logging 是关闭的，可以通过 behaviac.Config.IsLogging 和 behaviac.Config.IsSocketing 来分别打开或关闭 Logging 和 Connect。

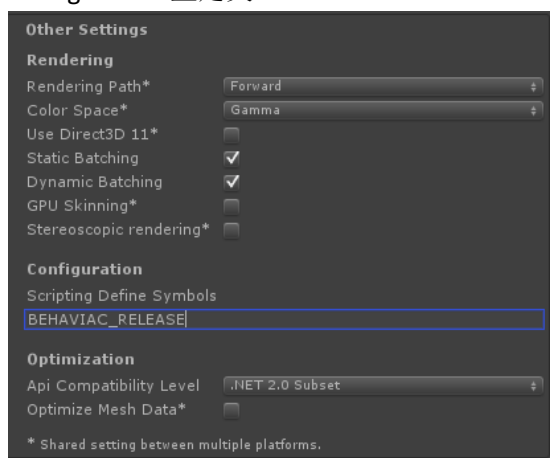
**但请注意**，当 Logging 或 Connect 打开的时候，效率有相当损失，也会产生相当的 GC Alloc。请参考 [13.8 关于 GC ALLOC 的说明](#)

当要对性能 profiling 的时候，或发布版本的时候，最好定义 BEHAVIAC\_RELEASE 来关闭 Logging 和 Connect 的支持：

可以在 smcs.rsp 中定义 BEHAVIAC\_RELEASE，如下图：



或者在 Unity Editor 里 Edit->Project Settings->Player，在 Inspector 里选择 Other Settings，在 Configuration 里定义：



需要说明的是，当没有定义 BEHAVIAC\_RELEASE 的时候，在 behaviac.Config.IsLogging 和 behaviac.Config.IsSocketing 都设置为 false 下，Logging 和 Connect 是关闭的，性能将基本 ok，虽然会有对 behaviac.Config.IsLoggingOrSocketing 是否为真的检测的少量开销。

## 13.6 Script 及设置

首先需要从创建一个 script，如下图：

```
public class Workspace : MonoBehaviour
{
    public string WorkspaceExportedPath;
    public behaviac.Workspace.EFileFormat FileFormat = behaviac.Workspace.EFileFormat.EFF_xml;
    //relative to the path of WorkspaceFile
    public string MetaPath;
    public bool bBlock;

    protected void Awake()
    {
        behaviac.Workspace.Instance.SetWorkspaceSettings(this.WorkspaceExportedPath, this.FileFormat);

        behaviac.Workspace.RegisterBehaviorNode();
        behaviac.Workspace.RegisterMetas();
    }

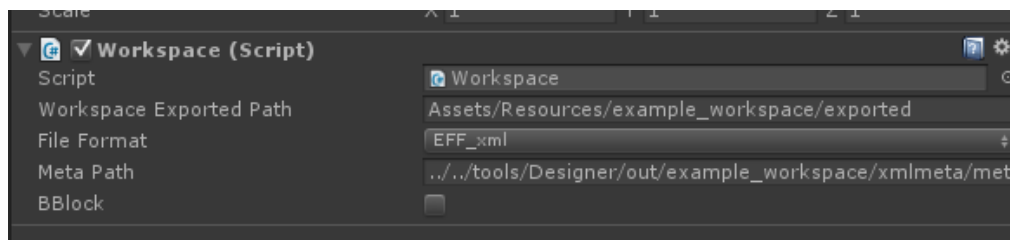
    protected void Start()
    {
        behaviac.Debug.Check(!string.IsNullOrEmpty(this.MetaPath));

        behaviac.Workspace.ExportMetas(this.MetaPath);

        behaviac.SocketUtils.SetupConnection(bBlock);
    }

    protected void Stop()
    {
        behaviac.SocketUtils.ShutdownConnection();
    }
}
```

在 unity 中将该 Workspace 添加到某个 GameObject 并且适当配置：



这里重要的主要是：

- 导出路径 ‘WorkspaceExportedPath’
- 所使用的BT的格式 ‘FileFormat’
- 元信息meta文件 ‘MetaPath’，这里是每次运行的时候都更新元信息meta文件，也可以考虑通过UI按钮更新或其他方式更新，请参考[元信息（meta），对Agent的描述](#)

需要指出的是，并非一定需要如上图那样创建自己的Workspace的Component，也可以在其他‘适当’的地方调用behaviac.Workspace的那几个函数即可。

然后，对于每一个需要控制的游戏对象，需要创建相应的 script，该 script 中的类需要从 behaviac.Agent 或子类继承，如下图，并且用相应的 Attribute 来修饰类型，属性，方法等，请参考 [C#中 Meta 的声明](#)：

```

[behaviac.TypeMetaInfo()]
public class AgentTestDoc : behaviac.Agent
{
    public AgentTestDoc()
    {
        action0_run = 0;
    }

    [behaviac.MethodMetaInfo()]
    public static int static_action2()
    {
        return s_count++;
    }

    [behaviac.EventMetaInfo()]
    delegate bool event_explode();

    [behaviac.MemberMetaInfo()]
    public int action0_run;

    [behaviac.MemberMetaInfo()]
    public int action1_run;

    [behaviac.MemberMetaInfo()]
    private int Property1;

    private static int s_count;

    protected void Awake()
    {
        behaviac.Debug.Log("PlayerTestBase Awake");

        base.Init();
    }

    protected void Start()
    {
        behaviac.Debug.Log("PlayerTestBase Start");

        this.btsetcurrent("CurrentBT");
    }

    protected void Update()
    {
        EBTStatus s = this.btexec();
        if (s != EBTStatus.BT_RUNNING)
        {
            this.btsetcurrent("CurrentBT");
        }
    }

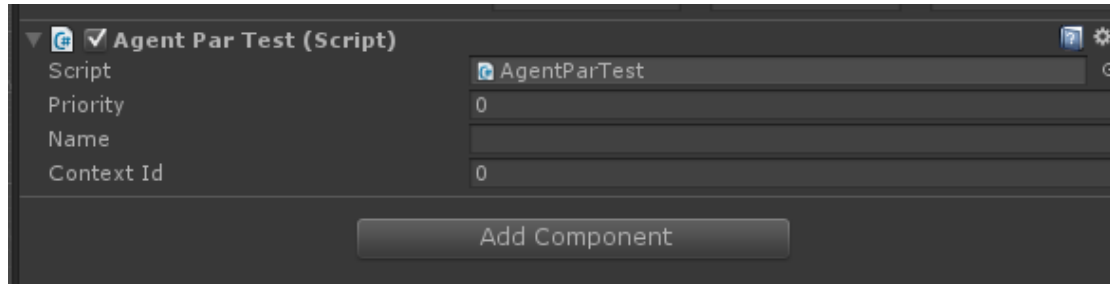
    protected void Stop()
    {
        behaviac.Debug.Log("PlayerTestBase Stop");
    }
}

```

请**特别注意**需要在 Awake 中或其他地方调用 base.Init(), 如果由于某种原因没有调用, 则在 <https://github.com/TencentOpen/behaviac>

执行 BT 的时候会有失败的 assert。

随后，把该 script 添加到相应的 GameObject，并且可以选择适当配置：



## 13.7 执行 BT

一般情况下，通过 `Agent.btload` 在初始化的时候来预加载 bt，通过 `Agent.btsetcurrent` 来设置‘当前’的 BT，通过 `Agent.btexec` 来执行当前的 BT。如果没有在初始化的时候调用 `Agent.btload` 预加载 bt，通过 `Agent.btsetcurrent` 设置当前 bt 的时候将会首先加载，而如果已经通过 `Agent.btload` 加载过的话，`Agent.btsetcurrent` 则直接使用之。

`Agent.btexec` 执行后总是返回一个状态表明执行结果：成功 `BT_SUCCESS`，失败 `BT_FAILURE`，或者运行 `BT_RUNNING`，如果没有‘当前’的 BT，则返回无效 `BT_INVALID`。一般情况下，如果返回的状态不是成运行 `BT_RUNNING`，则意味着‘当前’的 BT 结束了，需要调用 `btsetcurrent` 来重新设置 BT 做为‘当前’ BT。

## 13.8 关于 GC ALLOC 的说明

调试功能和 `System.Reflection.MethodBase.Invoke` 调用会产生 GC ALLOC。请参考 [13.5 Logging 和 Connect（连接 Designer）](#)。为了确保没有 GC ALLOC，需要：

- 定义 `BEHAVIAC_RELEASE`，关闭调试（Debug）功能
- 不能使用 xml 或 bson 格式（因为会调用 `System.Reflection.MethodBase.Invoke` 产生 GC ALLOC），使用 C# 格式的导出文件，并且指定使用 C# 而不是 xml 或 bson (`Workspace::SetWorkspaceSettings` 的第二个参数)
- Agent 中导出的函数都使用 `public`，请留意 unity editor 中的输出消息，如果不是 `public` 会有 warning 输出的。

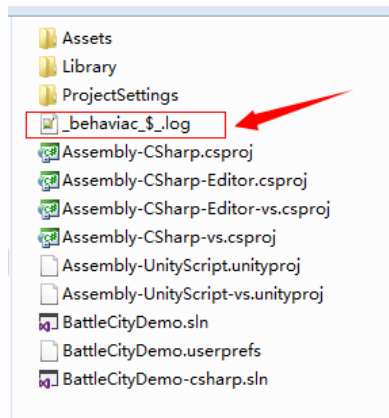
## 14 关于 log 文件的 FAQ

1. 去哪里看 log 文件？

运行时会产生一个 `_behaviac_$_.log` 的文件，关于运行状态的信息就被写到该文件里。

2. 这个 log 文件在哪里？

缺省情况下，该 log 文件产生在 exe 所在目录。Unity 下是在项目目录，即 Assets 所在目录。



3. 我可以修改 log 文件产生的目录吗？

可以调用 `LogManager::SetLogFilePath` 来设置 log 文件的路径。

4. 还是没有找到 log 文件！怎么回事？

缺省情况下 log 只在开发模式下打开，并且在 C# 下由 `behaviac.Config.IsLogging` 或 C++ 下由 `Behaviac::Config::IsLogging()` 控制是否打开。

- 在 C# 下，当 `BEHAVIAC_RELEASE` 没有定义的时候就是开发模式，请参考 [Logging 和 Connect（连接 Designer）](#)，还可以用 `behaviac.Config.IsLogging = true` 来打开或 `behaviac.Config.IsLogging=false` 来关掉 logging。
- 在 C++ 下，当 `BEHAVIAC_RELEASE` 没有定义的时候就是开发模式，请参考 [开发版本和最终版本](#)，还可以用 `behaviac::Config::SetLogging(true)` 来打开或 `behaviac::Config::SetLogging(false)` 来关掉 logging。

5. Log 时写入操作时很慢的操作，Flush 可以控制吗？

是的。可以用 `behavaic.LogManager.SetFlush()` 或 `behavaic::LogManager::SetFlush()` 来控制是否每次写入的时候都 Flush。

6. Log 文件的格式是什么？看不懂啊！

请参考 [Log 文件](#)，及 Inside Behaviac 中关于 Log 文件格式的说明。

## 15 和编辑器建立连接时的 FAQ

### 1. 游戏和编辑器不能建立连接？

Runtime 开发版才支持建立连接。在开发版下请检查是否调用了 `behaviac::Socket::SetupConnection`，请参考 [开发版本和最终版本](#)

另外，如果端口已经被占用，特别是已有的连接没有断开的情况下，或者已经有一个同样的游戏在运行，新的连接显然也不能建立。请确保只运行了一份该游戏。

此外，如果游戏没有正常退出，端口可能不被释放，则也会出现不能建立连接的情况，这种情况可能需要重启机器才能解决。

### 2. 游戏和编辑器建立连接后，没有 Agent 的实例？

- 请检查是否调用了 `SetIdMask` 和 `SetIdFlag`，请参考 [Agent::SetIdMask](#) 和 [Agent::SetIdFlag](#)

<https://github.com/TencentOpen/behaviac>

- 或者请确保游戏确实在更新。

### 3. 虽然有 Agent 的实例，但双击后没有 BT 树的更新显示？

如果不是调用 World::btexe 做的更新，需要调用

- Workspace::LogFrames 来 log 帧数
- Workspace::HandleRequests 来响应编辑器发过来的消息

如下图：

```
behaviac.Workspace.LogFrames ();  
behaviac.Workspace.HandleRequests ();
```

```
if(btloadResult)  
    btexec();
```

请参考[执行 BT, World::btexec](#)

### 4. 在 C#版本中，设置的断点似乎没被断下来？

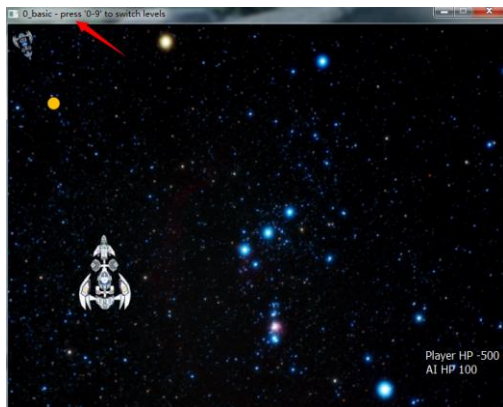
在 C#版本中，如果 behaviac.Workspace.RespondToBreakHandler 没有设置，则使用缺省的 MessageBox 来处理，但是该 MessageBox 仅在 WindowsEditor 和 WindowsPlayer 的版本有实现，在其他版本，特别是手机版本没有实现，由于没有窗口弹出，则用户感觉不到断点被断到。可以提供自己的函数实现并且添加到 RespondToBreakHandler。

## 16 Spaceship 的例子

安装包安装后再桌面会生成三个图标：



其中如红箭头所指的是 Launch 一个 spaceship 的 minigame。在安装目录里有所有的源码和项目文件，可以参考该目录里的 readme 获取更多信息。

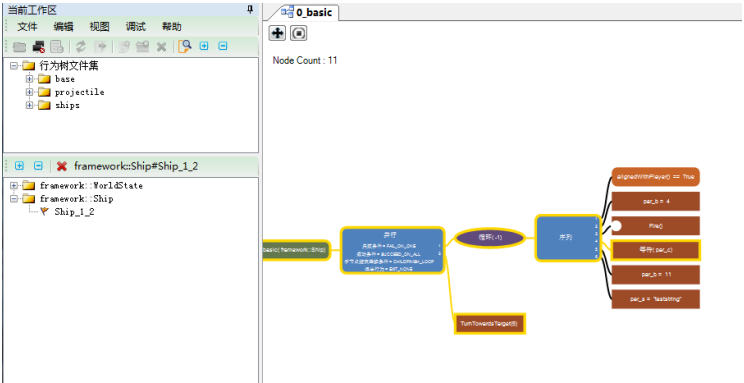


该小游戏运行的时候，可以通过按数字键 0 到 9 来选择 level。对应 level 中运行的 BT 树的

<https://github.com/TencentOpen/behaviac>

名字在窗口标题上有显示。游戏窗口中有两个 spaceship，下面的那个是由玩家控制，而上面的那个由 BT 控制。

运行游戏的时候，可以连接编辑器从而可以方便的看到 BT 更新的情况，对于学习相关节点的使用，掌握基本概念相信很有帮助。如下图，右下角列出 ship 的实例，需要首先双击实例从而打开该实例上激活的 BT，右侧就是该 BT 的更新情况，可以参考[检查 BT 的运行情况](#)。



而兰色箭头所指的是 unity 的 demo。

# 17 主要节点的介绍

Behaviac 有以下 5 类节点类型：

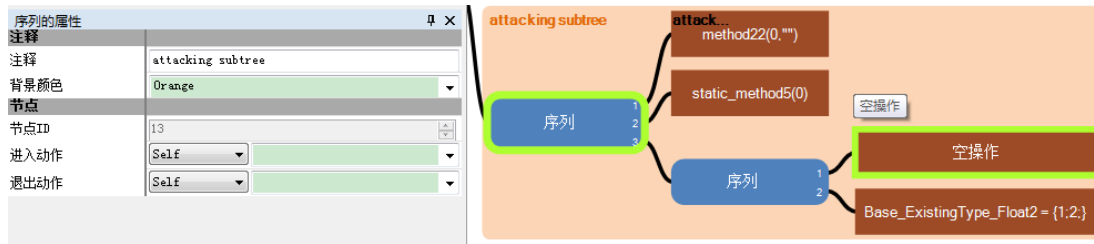


其中‘组合’，‘装饰器’类都是分支节点，‘动作’类，和‘条件’类大都是子节点，而‘条件’类中的‘或’、‘与’是分支节点。

需要指出的是，下面的有些节点类型通过其他更基本类型的节点的组合也可以实现。这里提供的节点类型只是使用起来可能会更方便，更直观些。用户可以根据自己的偏好选择使用与否。用户也可以选择扩展节点类型提供扩展功能或者快捷方式，请参考‘教程’中的相关章节。同时如果根本不打算使用某些节点类型，要可以在 config.xml 里配置后就不会出现在可选的列表里了，请参考[配置 Config.xml](#)。

## 17.1 节点共有属性

当在编辑器中选择每个节点的时候，都会有下图左侧所示的一些共有的属性。

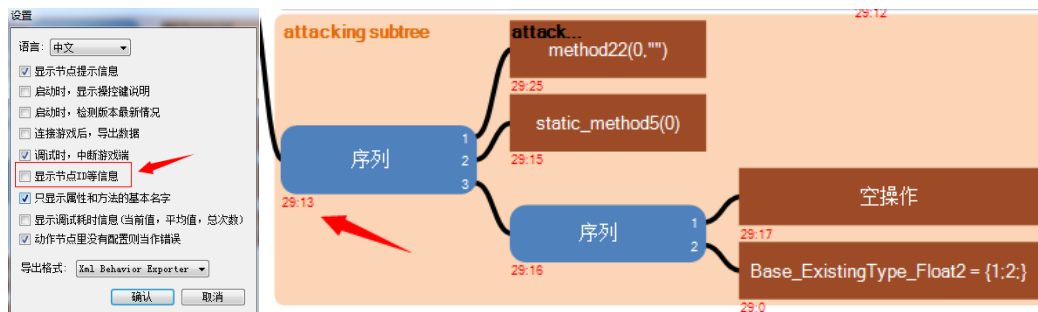


### 17.1.1 ‘注释’和‘背景颜色’

如上图右侧中所示的那样可以给某个部分表明一个颜色块和加上一些说明。

### 17.1.2 节点 ID

节点 ID 是节点的唯一 ID，可以用来定位节点，可以在文件->设置里打开“显示节点 ID 等信息”来在窗口里节点的左下角显示 ID。子树中的节点 ID 被加上主树节点的 ID，如下图。



进入动作，退出动作请参考[给节点配置 EnterAction/ExitAction](#)

## 17.2 节点状态

每个节点执行后都会返回下面状态之一：

- 成功 Success
- 失败 Failure
- 执行中 Running

在下面的说明中，涉及逻辑的时候，返回成功意味着就是返回 **true**，而返回失败意味着就是返回 **false**。理解这一点非常重要。

而返回成功或返回失败，意味着节点不再 Running，统称为节点结束。

执行一个 BT 的 exec 总是返回 EBTStatus 的这样的一个状态，成功、失败或运行；返回成功

<https://github.com/TencentOpen/behaviac>



可以认为是返回 **true**，返回失败可以认为是返回 **false**，有的时候认为返回 **true** 或 **false** 在上下文的理解上更方便些。

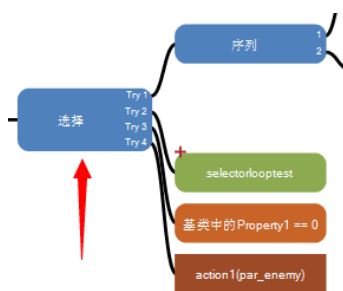
- 返回成功或失败意味着 BT 结束。下一次如果还被执行，则完全重新开始，即重新 **enter**，**update**，**exit**。
- 返回运行的情况则不同。如果本次执行返回运行，下次继续执行的时候，则继续执行（继续 **update**），如此循环直到返回成功或失败从而结束（**exit**）。特别的，该运行状态的节点在 BT 内部被‘记住’，下次执行的时候，‘直接’执行该运行状态的节点，而其他已经结束的节点不再被执行，理解这个处理对于把握 BT 的执行情况和效率比较**关键**。

由此可以看出 BT 就如同一个 **coroutine**，它 ‘知道’ 下一步从哪里继续。

## 17.3 组合节点 Composites

组合节点管理若干个子节点，也可以成为控制类节点。

### 17.3.1 选择（Selector）



**Selector** 节点是 BT 中传统的组合节点之一。该节点以给定的顺序依次调用其子节点，直到其中一个成功返回，那么该节点也返回成功。如果所有的子节点都失败，那么该节点也失败。

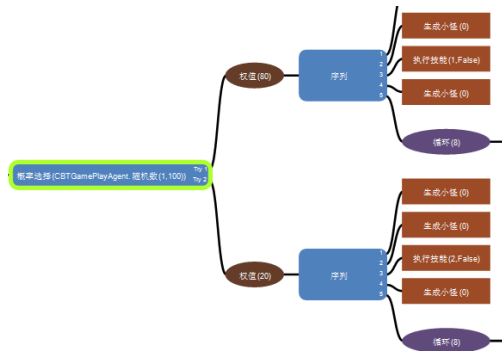
**Selector** 实现了 ‘||’ 的功能。我们知道表达式  $R=A||B||C||D$  执行的时候首先执行 A，如果 A 是 **true** 则返回 **true**，如果 A 是 **false** 则执行 B，如果 B 是 **true** 则返回 **true**，否则如果 B 是 **false** 则执行 C，如果 C 是 **true** 则返回 **true**，否则如果 C 是 **false** 则执行 D，并且返回 D 的值。

最一般的意义上，**Selector** 节点实现了一个**选择**，从子节点中选择一个。**Selector** 节点优先选择了排在前面的子节点。

此外，**Selector** 上还可以添加 **Predicate** 附件作为终止条件，请参考[判断（Predicate）](#)

具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\selector.cpp`

## 17.3.2 概率选择（SelectorProbability）



像 Selector 节点一样，SelectorProbability 也是从子节点中选择执行一个，不像 Selector 每次都是按照排列的先后顺序选择，SelectorProbability 每次选择的时候根据子节点的‘概率’选择，概率越大，被选到的机会越大。Selector 节点会顺序的去选择一直到成功的那个，而 SelectorProbability 的选择是‘直接’根据概率选择某个并且执行之，其他的则不会被执行。

特别需要指出的是，Selector 是按照子节点从上到下的顺序去执行子节点，一直到第一个返回成功的那个子节点然后返回成功，或者如果所有子节点都返回失败则返回失败。而 SelectorProbability 的不同之处则是，它根据概率，直接‘选择’某个子节点，执行之，无论其返回成功还是失败，SelectorProbability 也将返回同样的结果，当该子节点失败的话，SelectorProbability 也失败，它不会像 Selector 那样会继续执行接下来的子节点。

如下图，SelectorProbability 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。

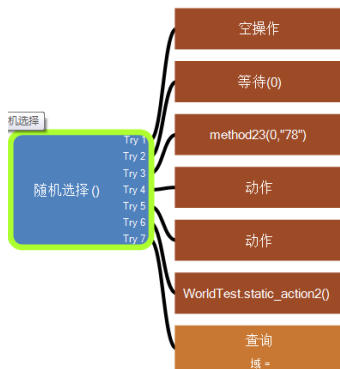


该随机数生成器也可以为空，则用系统的缺省实现。

SelectorProbability 的子节点只能是‘权值’的子节点，该被节点系统自动添加。所有子节点的权值相加之和 sum 不需要是 100，子节点的概率是该子节点的（权值/sum）。

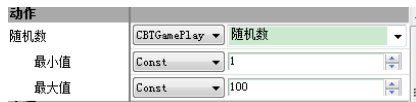
具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\selectorprobability.cpp`

### 17.3.3 随机选择（SelectorStochastic）



像 Selector 节点一样，SelectorStochastic 也是从子节点中选择执行一个，不像 Selector 每次都是按照排列的先后顺序选择，SelectorStochastic 每次选择的时候随机的决定执行顺序。假如 Selector 和 SelectorStochastic 都有 A, B, C, D 子节点。对于 Selector，每次都是顺序的按 A, B, C, D 的顺序选择，而对于 SelectorStochastic，有时按 A, B, C, D 的顺序选择，有时按 B, A, D, C 的顺序选择，又有时按 A, C, D, B 的顺序选择，等等。

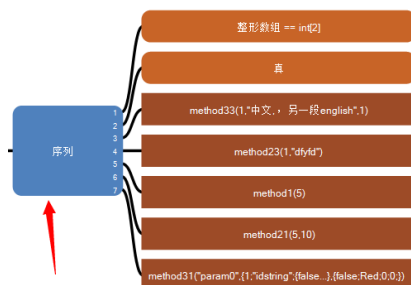
和 SelectorProbability 相同的是，如下图，SelectorStochastic 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。



该随机数生成器也可以为空，则用系统的缺省实现。

具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\selectorstochastic.cpp`

### 17.3.4 序列（Sequence）



Sequence 节点是 BT 中传统的组合节点之一。该节点以给定的顺序依次执行其子节点，直到所有子节点成功返回，该节点也返回成功。只要其中某个子节点失败，那么该节点也失败。

Sequencer 实现了‘&&’的功能。我们知道表达式 `R=A&&B&&C&&D` 执行的时候首先执行 A，

<https://github.com/TencentOpen/behaviac>

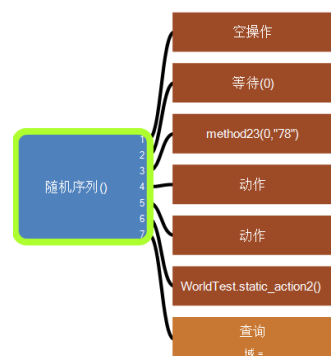
如果 A 是 false 则返回 false，如果 A 是 true 则执行 B，如果 B 是 false 则返回 false，否则如果 B 是 true 则执行 C，如果 C 是 false 则返回 false，否则如果 C 是 true 则执行 D，并且返回 D 的值。

最一般的意义上，Sequence 节点实现了一个序列。实际上，Sequence 节点不仅可以管理‘动作’子节点，也可以管理‘条件’子节点。如上图的应用中，首先两个条件节点，跟着若干个其他节点，这两个条件节点实际上用作进入下面其他节点的 precondition，只有这两个条件是 true，下面的其他节点才有可能执行。

此外，Sequence 上还可以添加 Predicate 附件作为终止条件，请参考[判断（Predicate）](#)

具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\sequence.cpp`

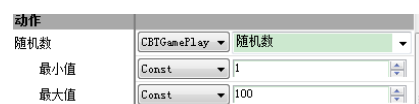
### 17.3.5 随机序列（SequenceStochastic）



像 Sequence 节点一样，SequenceStochastic 也是从子节点中顺序执行，不像 Sequence 每次都是按照排列的先后顺序，SequenceStochastic 每次执行的时候随机的决定执行顺序。

假如 Sequence 和 SequenceStochastic 都有 A, B, C, D 子节点。对于 Sequence，每次都是顺序的按 A, B, C, D 的序列，而对于 SequenceStochastic，有时按 A, B, C, D 的序列，有时按 B, A, D, C 的序列，又有时按 A, C, D, B 的序列，等等。

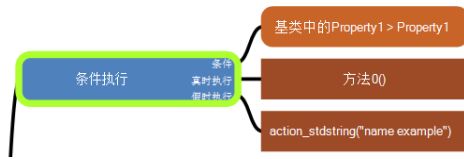
和 SelectorStochastic 相同的是，如下图，SequenceStochastic 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。



该随机数生成器也可以为空，则用系统的缺省实现。

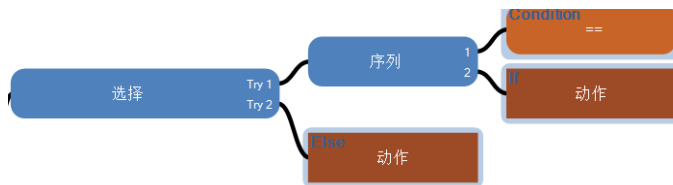
具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\sequencestochastic.cpp`

## 17.3.6 条件执行（IfElse）

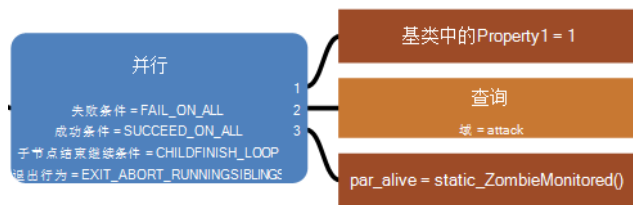


IfElse 是 behaviac 的一个扩展或快捷方式。IfElse 一定有 3 个子节点，第一个子节点是条件节点，第二个子节点是 If 分支，第三个子节点是 Else 分支。如果条件为真，那么执行 “If” 分支；否则，执行 “Else” 分支。而 IfElse 的执行结果则根据具体执行分支的执行结果来决定。

如果不使用 IfElse，完全可以用 Sequence 和 Selector 实现相同的功能，只不过显得有些臃肿。例如：



## 17.3.7 并行（Parallel）



Parallel 节点在最一般意义上是并行的执行其子节点。在 Selector 和 Sequence 中，‘顺序’的‘一个一个’的执行子节点，上一个子节点执行结束后，根据其状态是否执行接下来的子节点。Parallel 节点在逻辑上是‘同时’并行的执行所有子节点，然后根据所有子节点的状态决定本身的状态。如何根据所有子节点的状态决定本身的状态呢？具体的如下图，Parallel 节点有几个属性可以配置：

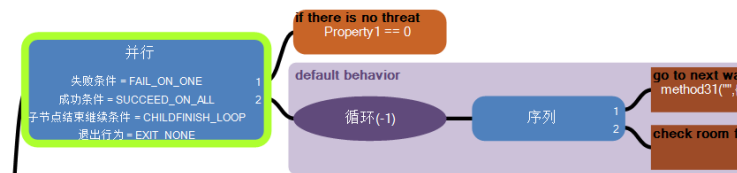
并行	
失败条件	FAIL_ON_ALL
成功条件	SUCCEED_ON_ALL
子节点结束继续条件	CHILDFINISH_LOOP
退出行为	EXIT_ABORT_RUNNINGSIBLINGS

- 失败条件，决定 Parallel 节点在什么条件下是失败的
- 成功条件，决定 Parallel 节点在什么条件下是成功的
- 子节点结束继续条件，子节点结束后是重新再循环执行还是结束后不再执行
- 退出行为，当 Parall 节点的成功条件或失败条件满足而成功或失败后，是否需要 abort 掉其他还在运行的子节点

- 当子节点执行状态既不满足失败条件，也不满足成功条件，且无 Running 状态子节点时，Parallel 节点返回 Failure

在序列 Sequence 中，当条件节点之后跟着其他节点的时候，条件节点实际上是作为其他节点的 precondition，只有条件节点是 true，下面的其他节点才有可能执行。

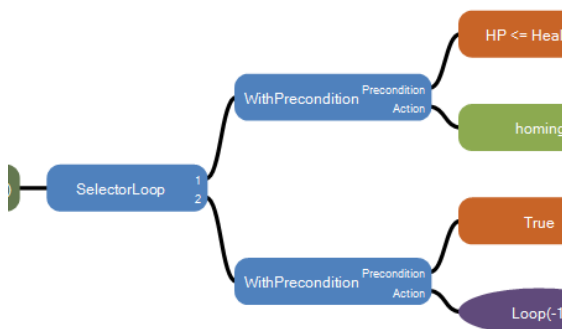
Parallel 节点可以用来实现所谓 ‘context precondition’，如下图：



Parallel 节点配置条件节点和其他节点，并且失败条件是缺省的配置 FAIL\_ON\_ONE，那么只有当条件节点成功的时候其他节点才被执行，从而条件节点事实上是其他节点的 precondition。和 Sequence 的不同之处在于，Sequence 节点实现的 precondition 只是 ‘进入’ 其他节点的 precondition，一旦 ‘进入’ 开始执行其他节点就不再检查该 precondition 了，而 Parallel 节点实现的 precondition 是 ‘context’ 的，不但 ‘进入’ 开始执行前需要检查，之后每次执行也都要检查。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\parallel.cpp

### 17.3.8 循环选择（SelectorLoop）和条件动作（WithPrecondition）



SelectorLoop 和 WithPrecondition 作为对传统 BT 的扩展，对于处理事件比较直观和方便。

SelectorLoop 和 WithPrecondition 只能配对使用，即 SelectorLoop 只能添加 WithPrecondition 作为它的子节点，WithPrecondition 也只能作为 SelectorLoop 的子节点被添加。

- WithPrecondition 有 precondition 子树和 action 子树。只有 precondition 子树返回 success 的时候，action 子树才能够被执行。
- SelectorLoop 是一个动态的选择节点，和 Selector 相同的是，它选择第一个 success 的节点，但不同的是，它不是只选择一次，而是每次执行的时候都对其子节点进行选择。如上图中，假若它选择了下面有 True 条件的那个节点并且下面的 Loop 节点在运行状态，

<https://github.com/TencentOpen/behaviac>

下一次它执行的时候，它依然会去检查上面的那个有条件  $HP \leq \text{Health}$  的子树，如果该条件为真，则终止下面的运行节点而执行上面的 homing 子树。

- 当 Action 子树返回成功（success）的时候，SelectorLoop 节点也将结束并且返回成功，但是需要指出的是，当 Action 子树返回失败（failure）的时候，SelectorLoop 节点将继续尝试接下来的节点！

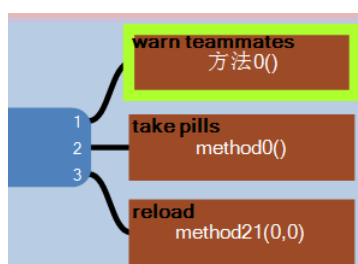
请参考 [SelectorLoop](#) 和 [WithPrecondition](#)

具体的执行逻辑可以查看 `src\behaviortree\nodes\composites\selectorloop.cpp`

## 17.4 叶子节点

叶子节点只能作为最底层的叶子加到 BT 上去。不像组合节点是作为分支节点来控制其他节点。叶子节点必然被加到了其他组合节点上。

### 17.4.1 动作（Action）



Action 节点是 behaviac 中几乎是最重要的节点。Action 节点通常对应 Agent 的某个方法（Method），可以从下拉列表里为其选择方法。在设置其方法后，需进一步设置其“决定状态的选项（Status Option）”或“决定状态的函数（Status Functor）”，如下图所示，如果没有正确配置，则视为错误不可能被导出：

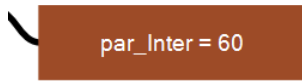


请参考[动作（Action）](#)

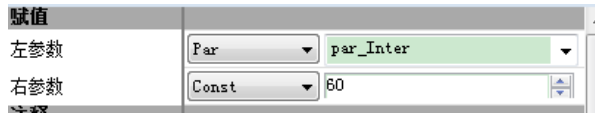
此外，Action 上还可以添加 Predicate 附件作为 Precondition，请参考[判断（Predicate）](#)

具体的执行逻辑可以查看 `src\behaviortree\nodes\actions\action.cpp`

## 17.4.2 赋值（Assignment）



Assignment 节点实现了一个赋值的操作，可以把右边的值赋值给左侧的参数。



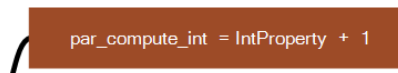
其中左参数可以是 par 或者是某个 agent 的属性，右参数可以是常数、par，其他 agent 的属性、或者是方法调用的返回值。

当需要对某个属性或 par 做一些加减乘除运算的时候，可以用节点[计算（Compute）](#)。但请注意这些操作的‘粒度’过小，大量这种小‘粒度’的操作可能对性能造成影响。请慎用。

另外如果需要修改某些其他没有导出的属性，或做一些复杂的计算时，可以通过 Action 节点调用相应的函数类实现修改或计算。

具体的执行逻辑可以查看 `src\behaviortree\nodes\actions\assignment.cpp`

## 17.4.3 计算（Compute）



Compute 节点对属性，par 或函数的返回值做加减乘除的运算，把结果赋值给某个属性或 par。



其中左参数可以是 par 或者是某个 agent 的属性，参数 1 和参数 2 可以是常数、par，其他 agent 的属性、或者是方法调用的返回值。操作符可以是“+,-,\*,/”。

另外如果需要修改某些其他没有导出的属性，或做一些复杂的计算时，可以通过 Action 节点调用相应的函数类实现修改或计算。

但请注意这些操作的‘粒度’过小，大量这种小‘粒度’的操作可能对性能造成影响。请慎用。



## 17.4.4 等待（Wait）

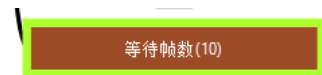


Wait 节点保持在 Running 状态持续在指定的时间（毫秒 ms），之后返回成功。

需要配置‘持续时间’，可以是常数，也可以是 par 或属性。



## 17.4.5 等待帧数（WaitFrames）

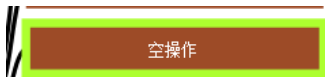


WaitFrames 节点保持在 Running 状态持续在指定的帧数，之后返回成功。

需要配置‘帧数’，可以是常数，也可以是 par 或属性。



## 17.4.6 空操作（Noop）



Noop 节点只是作为占位，仅执行一次就返回成功。

## 17.4.7 查询（Query）



Query 节点是 behaviac 的扩展，大部分情况下请忽略。

Query 节点试图提供一个动态的功能，例如，Query 节点可以用作如下场景：

当我们需要一个用于‘attack’的子树，而用于攻击的子树在创建拥有该 Query 节点的 BT 的时候可能还没有创建，可以配置该 Query 节点的 domain 为‘attack’，而所有需要用作‘attack’的 BT 的 domains 都配置上‘attack’，这样在执行的时候，Query 节点就从所有 domains 出现了‘attack’的 BT 中挑选，然后按照配置的描述器来进行排序从而选择最合适的那个。

具体工作原理如下：

<https://github.com/TencentOpen/behaviac>

Query 节点有如下两个属性：

查询	
域	attack
描述器	Descriptor_t[3]

- 域 domain 是个字符串，用来表明类似标签一样的某种目的，比如分类，比如特征等。这里只是一个字符串。
- 描述器是个数组，每个元素是如下的一个结构，属性是某个 agent 的属性，引用是相应类型的参考值，权重是个百分比。

描述器的属性	
属性	test_node.P Property1
引用	Const
权重	10.0
数值 权重	
关闭	

而每个 BT 也有如下的两个相关属性：

查询	
域	attack defence
DescriptionRefs	DescriptorRef[1]

- 域 domains 是个字符串，用来表明类似标签一样的某种目的，比如分类，比如特征等。这里可以是多个空格分隔的字符串，表明该 BT 的某个特征，比如分类。
- DescriptionRefs 是参考值得数组。每个元素如下图：

DescriptorRefs的属性	
描述器	Descriptor
引用	Const
数值 权重	
关闭	

Runtime 的执行逻辑如下图，对于每一个加载的 BT，查看 Query 节点中配置的 domain 是否出现在该 BT 的 domains 中，如果出现了，则对该 BT 用配置的描述器计算一个相似度，最后选择那个最相似的 BT 来执行。

```

const Query* pQueryNode = Query::DynamicCast(this->GetNode());
if (pQueryNode)
{
    const Query::Descriptors_t& qd = pQueryNode->GetDescriptors();
    if (qd.size() > 0)
    {
        const Workspace::BehaviorTrees_t& bs = Workspace::GetBehaviorTrees();

        BehaviorTree* btFound = 0;
        float similarityMax = -1.0f;

        for (Workspace::BehaviorTrees_t::const_iterator it = bs.begin();
             it != bs.end(); ++it)
        {
            BehaviorTree* bt = it->second;

            const behaviorac::string_t& domains = bt->GetDomains();

            if (pQueryNode->m_domain.empty() || domains.find(pQueryNode->m_domain) != behaviorac::string_t::npos)
            {
                const BehaviorTree::Descriptors_t& bd = bt->GetDescriptors();

                float similarity = pQueryNode->ComputeSimilarity(qd, bd);

                if (similarity > similarityMax)
                {
                    similarityMax = similarity;
                    btFound = bt;
                }
            }
        }

        if (btFound)
        {
            //BehaviorTreeTask* pAgentCurrentBT = pAgent->btGetCurrent();
            //if (pAgentCurrentBT && pAgentCurrentBT->GetName() != btFound->GetName())
            {

                const char* pReferencedTree = btFound->GetName().c_str();
                pAgent->btsetCurrent(pReferencedTree, TM_Return);

                return true;
            }
        }
    }
}

```

相似度是通过下述的代码计算获得，也就是对配置在 Query 节点上的每一个属性在 BT 上查找是否也配置了，如果配置了则比较在 Query 节点上的配置的参考值和 BT 上配置的参考值以及其权值得到一个数值即相似度。

```

const Property* Query::FindProperty(const Query::Descriptor_t& q, const BehaviorTree::Descriptors_t& c)
{
    BehaviorTree::Descriptors_t::const_iterator it = std::find_if(c.begin(), c.end(), PropertyFinder_t(q));
    if (it != c.end())
    {
        return it->Descriptor;
    }

    return 0;
}

float Query::ComputeSimilarity(const Query::Descriptors_t& q, const BehaviorTree::Descriptors_t& c) const
{
    float similarity = 0.0f;
    for (size_t i = 0; i < q.size(); ++i)
    {
        const Descriptor_t& qi = q[i];

        const Property* ci = FindProperty(qi, c);

        if (ci)
        {
            float dp = qi.Attribute->DifferencePercentage(ci);

            BEHAVIAC_ASSERT(dp >= 0.0f && dp <= 1.0f, "dp should be normalized to [0, 1], please check its scale");

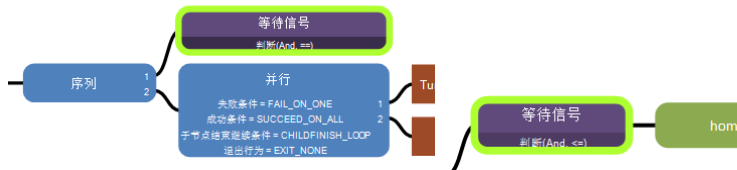
            similarity += (1.0f - dp) * qi.Weight;
        }
    }
}

```

Query 节点上还可以添加若干判断 Predicate 附件作为重新 Query 的条件。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\query.cpp

## 17.4.8 等待信号（WaitforSignal）



WaitforSignal 节点模拟了一个等待某个条件的‘阻塞’过程。在上左图所示的情况下，该 WaitforSignal 节点‘阻塞’直到它上面附加的 Predicate 判断的条件是 true 的时候才结束‘阻塞’，从而继续序列中后面的节点。而如上右图中，则是它上面附加的 Predicate 判断的条件是 true 的时候，结束‘阻塞’而执行其子节点。

WaitforSignal 节点返回 Running，一直到它上面附加的 Predicate 判断的条件是 true 的时候，如果有子节点，则执行其子节点并当子节点结束的时候返回子节点的返回值，如果没有子节点则返回成功。

具体的执行逻辑可以查看 `src\behaviortree\nodes\actions\waitforsignal.cpp`

## 17.5 条件节点

条件节点不会返回 Running，条件节点的执行被认为是“即时”的，一个 exec 的执行要么返回成功 Success，要么返回 Failure，返回成功的时候视作 true，返回失败的时候视作 false。

### 17.5.1 条件（Condition）

par\_CurPeriodIdx <= 2

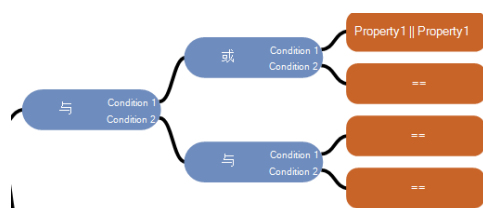
Condition 节点对左右参数进行比较，如果结果为 True，返回成功，如果结果为 False，返回失败，Condition 节点不可能返回 running。通常左参数是 par、Agent 的某个属性或 Agent 某个有返回值方法的调用，用户可以从下拉列表里选择，右参数是相应类型的常数、par 或 Agent 的某个属性。

Condition 节点没有提供取反的属性。如果需要取反，请用装饰节 [非 \(Not\)](#)。

参考 [条件 \(Condition\)](#)

具体的执行逻辑可以查看 `src\behaviortree\nodes\conditions\condition.cpp`

## 17.5.2 或（Or）及与（And）



Or 节点接受两个条件子节点，执行一个逻辑或，只要一个条件子节点返回值为成功（true），则返回成功，两个条件子节点都返回为失败（false）的时候返回失败（false）。  
And 节点接受两个条件子节点，执行一个逻辑与，只要一个条件子节点返回值为失败（false），则返回失败，两个条件子节点都返回为成功（true）的时候返回成功（true）。

## 17.5.3 真（True）及假（False）



True 节点和 False 节点往往也是作为占位。在某些需要一个 Condition 的时候，可以拿 True 或 False 来占位或测试。

True 节点总是返回成功。False 节点总是返回失败。

## 17.6 装饰节点

装饰节点作为控制分支节点，必须且只接受一个子节点。装饰节点的执行首先执行子节点，根据自身的控制逻辑以及子节点的返回结果决定自身的状态。

装饰节点都有属性 `DecorateChildEnds` 可以配置：

基本	
子节点结束时作用	<input type="checkbox"/>

如果 `DecorateChildEnds` 配置为 `true`，则仅当子节点结束（成功或失败）的时候，装饰节点的装饰逻辑才起作用。

### 17.6.1 输出消息（Log）



执行完子节点后，输出配置的消息。Log 节点可以作为调试的辅助工具。

## 17.6.2 非 (Not)



该装饰器将子节点的返回值取反。如果子节点失败，那么此节点返回成功。如果子节点成功，那么此节点返回失败。如果子节点返回 Running，则同样返回 Running。

具体的执行逻辑可以查看 `src\behaviortree\nodes\decorators\decoratornot.cpp`

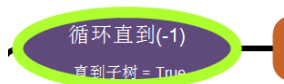
## 17.6.3 循环 (Loop)



Loop 节点循环执行子节点指定的次数。当指定的次数到达后返回成功。在指定的次数到达前一直返回 Running。如果指定的次数是-1 则无限循环，总是返回 Running。

具体的执行逻辑可以查看 `src\behaviortree\nodes\decorators\decoratorloop.cpp`

## 17.6.4 循环直到 (LoopUntil)



LoopUntil 节点除了像 Loop 节点可以配置循环的次数，还有一个属性‘直到子树’需要配置。



LoopUntil 节点有两个结束条件，指定的‘循环次数’到达或子树返回值和‘直到子树’值一样，只要一个条件满足就结束。

- 指定的‘循环次数’到达的时候，返回成功。
- 指定的‘循环次数’是-1 的时候，是无限循环，等同于只检查子树的返回值是否满足。
- 子树的返回值满足的时候：
  - 当‘直到子树=true’的时候，意味着直到子树返回成功，也返回成功。
  - 当‘直到子树=false’的时候，意味着直到子树返回失败，也返回失败。

具体的执行逻辑可以查看 `src\behaviortree\nodes\decorators\decoratorloopuntil.cpp`

## 17.6.5 计数限制 (CountLimit)



<https://github.com/TencentOpen/behaviac>

CountLimit 节点不同于循环节点。CountLimit 节点在指定的循环次数到达前返回子节点返回的状态，无论成功失败还是 Running。

在指定的循环次数到达后不再执行，CountLimit 节点的 onenter 直接返回 false。

如果指定的循环次数是-1，则无限循环，等同于什么操作都没有，只是执行子节点并且返回子节点的返回值。

此外，CountLimit 上还可以添加 Predicate 附件作为重新开始条件，请参考[判断 \(Predicate\)](#)

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorcountlimit.cpp

## 17.6.6 返回成功直到 (FailureUntil) / 返回失败直到 (SuccessUntil)



FailureUntil 节点在指定的次数到达前返回失败，指定的次数到达后返回成功。如果指定的次数是-1，则总是返回失败。

SuccessUntil 节点在指定的次数到达前返回成功，指定的次数到达后返回失败。如果指定的次数是-1，则总是返回成功。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorfailureuntil.cpp 或 src\behaviortree\nodes\decorators\decoratorsuccessuntil.cpp

## 17.6.7 总是成功 (AlwaysSuccess) / 总是失败 (AlwaysFailure) / 总是运行 (AlwaysRunning)



AlwaysSuccess 节点无论子节点返回什么，它总是返回成功。

AlwaysFailure 节点无论子节点返回什么，它总是返回失败。

AlwaysRunning 节点无论子节点返回什么，它总是返回 Running。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratoralwaysuccess.cpp 或 src\behaviortree\nodes\decorators\decoratoralwaysfailure.cpp 或 src\behaviortree\nodes\decorators\decoratoralwaysrunning.cpp

## 17.6.8 时间（Time）及帧数（Frames）



在指定的时间内或帧数内执行子节点，返回 Running，当超出了指定的时间或帧数后，则返回成功。

具体的执行逻辑可以查看 `src\behaviortree\nodes\decorators\decoratortime.cpp` 或 `src\behaviortree\nodes\decorators\decoratorframes.cpp`

## 17.7 附件

附件类节点只能作为‘附件’添加到其他节点上。附件类节点不可能独立的添加到 BT 上去。

### 17.7.1 判断（Predicate）



Predicate 附件的功能就像是 Condition 节点，它的功能是做出比较，实际是个条件判断，返回成功（true）或失败（false）。

在节点类的代码里，可以提供下面的函数 `AcceptsAttachment` 来决定是否可以接受某类附件：  
`public override bool AcceptsAttachment(Type type)`

如下图，每个 Predicate 附件有个 Association 用来表明本 Predicate 是如何和后续的 Predicate 共同起作用的。

Association	And
Left	Self
Operator	==
Right	Const

需要指出的是，多个 Predicate 运算的时候，总是先计算最左（前面）的那个 Predicate，

如：P1 And P2 Or P3，首先计算 P1，

如果 P1=true，由于接下来的是 And，则计算 P2，

如果 P2 也是 true，则 (P1 And P2) = true 并且接下来的是 Or，则直接返回 true。

如果 P2 也是 false，则 (P1 And P2) = false，由于接下的是 Or，则计算 P3，如果 P3=true，则返回 true，否则返回 false。

如果 P1=false，由于接下来的是 And，则直接返回 false

目前 Action, Sequence, Selector, Query, CountLimit 和 WaitForSignal 可以接受 Predicate 附件，

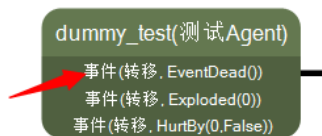
<https://github.com/TencentOpen/behaviac>



其他类型的节点暂不接受。

- 对于 Action 节点, 附加的 Predicate 附件作为 precondition 的条件。只有当该 precondition 条件为 true 的时候, Action 节点配置的 Method 才会执行, 如果该 precondition 条件为 false 的时候, Action 节点失败。特别需要说明的是, 该 precondition 是**每次**执行 method 之前都会首先被执行。
- 对于 Sequence 和 Selector 节点, 附加的 Predicate 附件作为打断序列的条件, 当顺序的执行完一个子节点后则检查附加的 Predicate 节点, 如果是 false, 则返回失败, 不再继续后面的子节点。
- 对于 Query 节点, 附加的 Predicate 附件作为 ReQuery 的条件。
- 对于 CountLimit 节点, 附加的 Predicate 附件作为重新计数的条件。
- 对于 WaitForSignal 节点, 附加的 Predicate 附件作为‘阻塞’执行的条件。

## 17.7.2 事件 (Event)



用鼠标拖拽 BT 到节点就可以添加 Event。当添加 Event 的时候, BT 的 Agent 类型需要保持‘一致’。如下图, 可以用鼠标拖拽 dummy\_test 到打开的某个 BT 窗口内的某个节点上添加 Event。



请参考 [Event 作为附件](#)

## 18 Revision

2014 年 8 月 28 日

[和编辑器建立连接时的 FAQ](#)

2014 年 8 月 20 日

[关于 char, signed char, unsigned char 等的说明](#)

2014 年 7 月 2 日

[Logging 和 Connect \(连接 Designer\)](#)

[Editor Script](#)

[关于 log 文件的 FAQ](#)

[导出元信息](#)

<https://github.com/TencentOpen/behaviac>

2014 年 5 月 13 日

[Behaviac 项目概况](#)

[Log 文件](#)

[执行 BT，World::btexec](#)

[FileManager](#)

[Unity 中使用 behaviac](#)

[C#中 Meta 的声明](#)

[导出路径](#)

[主要节点的介绍](#)

2013 年 10 月 28 日

关于导出

2013 年 3 月 29 日

初始版本